GÉNÉRATION DE TESTS DE VULNÉRABILITÉ POUR DES PROGRAMMES JAVA CARD ITÉRATIFS

par

Vincent Fély

Mémoire présenté au Département d'informatique
en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES

UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, 14 août 2019

Le 14 août 2019

Le jury a accepté le mémoire de Monsieur Vincent Fély dans sa version finale.

Membres du jury

Professeur Marc Frappier
Directeur de recherche
Département d'informatique

Professeur Michael Blondin
Évaluateur interne au programme
Département d'informatique

Professeur Manuel Lafond
Président-rapporteur
Département d'informatique

« Life is too short for manual testing. »

Harry Robinson, test architect for Microsoft's Engineering Excellence Group and a driving force behind Microsoft's model-based testing initiative, 1999–2004.

Sommaire

Utilisée très largement par le système bancaire actuel, la carte à puce peut être considérée comme un système critique. La majorité de ces cartes évoluent sous le système d'exploitation pour carte à puce Java Card, dont le code source est fermé. Leur évaluation repose donc sur des méthodes de tests en boîte noire, c'est-à-dire des méthodes dans lesquelles les tests générés ne se basent que sur la spécification fournie par le propriétaire du système sous test. Il existe différentes méthodes pour générer des tests en boîte noire mais l'intervention humaine nécessaire pour les concevoir est importante pour une couverture très souvent limitée.

Des travaux ont donc été entrepris ces dernières années pour répondre à ces deux problématiques et ont abouti à la création de la méthode Vulnerability Test Generation (VTG). Appliquée à l'évaluation du vérifieur de code intermédiaire de Java Card (VCI), cette méthode a permis de générer de manière automatique des tests d'une grande qualité dans un temps réduit.

Les travaux présentés dans ce mémoire ont permis d'aller plus loin dans l'étude de cette méthode en proposant de nouvelles approches et méthodologies. Nous avons ainsi pu expérimenter l'utilisation de différentes méthodes formelles de manière à établir un comparatif sur des critères de performance, d'adaptabilité et de coût en temps humain. Basés sur ces résultats, nous avons étendu le cas d'étude du VCI en modélisant de nouvelles instructions. Ces avancées ont rendu possible une évaluation plus complète des mécanismes de vérification du VCI.

Mots-clés: VTG ; tests de vulnérabilité ; Java Card ; Event-B ; ProB ; model-based testing ; mutation testing

Remerciements

Au professeur Marc Frappier, pour m'avoir offert l'opportunité de rejoindre son équipe et de travailler à ses côtés pendant ces quatre années. Merci à lui de m'avoir initié au monde des méthodes formelles.

À Aymerick Savary, pour tous ces moments passés à me former sur la méthode et sur le cas d'étude. Merci à lui pour tout le soutien qu'il m'a apporté tout au long du projet.

À la professeure Régine Laleau, pour m'avoir accueilli au LACL pendant l'été 2016. Merci également à Sergiu, Martin et Yackolley pour tous ces échanges et ces soirées passées au laboratoire.

À l'Université de Limoges et notamment à Benoît Crespin, pour avoir rendu possible cet échange avec l'Université de Sherbrooke. Merci à lui pour m'avoir apporté toutes les ressources nécessaires à la réussite de cette fabuleuse aventure.

À mes parents, pour m'avoir transmis très tôt leur amour de la science, et à ma merveilleuse compagne, pour me permettre de le cultiver en m'apportant chaque jour son soutien sans failles.

À Alexandre Guertin, ~~last but not least~~ dernier mais non des moindres, pour m'avoir fait découvrir le Québec - libre ! -, sa culture, sa gastronomie et sa chaleur - même l'hiver quand *y fait frette en tabarouette*.

Remerciements

Abréviations

API Application Programming Interface

CAP Converted APplet

CBC Constraint Based Checking

JCA Java Card Assembly

JCVM Java Card Virtual Machine

JVM Java Virtual Machine

MBT Model-Based Testing

SUT System Under Test

VTG Vulnerability Tests Generator / Vulnerability Tests Generation

VCI Vérifieur de Code Intermédiaire

XML eXtensible Markup Language

Table des matières

Liste des figures

Liste des tableaux

Liste des tableaux

Liste des programmes

Introduction

Contexte

La carte puce est née du besoin de rendre transportable de faibles quantités d'informations tout en leur assurant une protection forte contre les accès non-autorisés. Utilisée aujourd'hui aussi bien comme moyen de paiement que comme porteuse de l'identité d'une personne, elle renferme des données toujours plus sensibles et convoitées.

La majorité des cartes à puces utilisent la technologie Java Card qui met à disposition un environnement permettant de programmer, de charger puis d'exécuter sur une carte à puce des applications préalablement contrôlées par des mécanismes de vérification. L'un d'eux, le «*Java Card Byte Code Verifier*», ou VCI (Vérifieur de Code Intermédiaire), assure l'intégrité et la validité d'une application en effectuant des vérifications sur le code de plus bas niveau d'une application Java Card, aussi appelé «*byte code*» (code intermédiaire). Une défaillance sur le VCI pourrait donc potentiellement conduire à une mise en défaut de la carte à puce dans le cas d'une attaque logique, c'est-à-dire une attaque exploitant des défaillances au niveau des logiciels.

L'une des méthodes permettant la vérification de la sûreté d'un système est la génération de tests. Le VCI est un système propriétaire fermé. Il n'est donc pas possible pour des personnes extérieures à l'entreprise propriétaire de consulter son code source pour l'évaluer. Il faut donc employer des techniques de tests en boîte noire, c'est-à-dire des tests conçus uniquement sur la base de la spécification fournie par le propriétaire du système. Parmi ces techniques, le MBT («*Model-Based Testing*», test à base de modèle) permet de générer des tests à partir d'une modélisation abstraite du

système sous test, ou SUT («*System Under Test*»). La méthode VTG («*Vulnerability Test Generation*», génération de tests de vulnérabilité), développée par Savary et al. [15], a fait l'objet de nombreuses publications ces dernières années. Elle combine le MBT avec de la mutation de spécification pour générer de manière automatique des tests de vulnérabilité à partir d'une spécification fonctionnelle. C'est sur cette dernière méthode que porte notre sujet d'étude.

Problématique

Les travaux précédemment entrepris sur la méthode VTG [2, 15] ont permis de prouver qu'il est possible d'utiliser cette méthode pour générer de manière automatique des tests de vulnérabilité pour vérifier le VCI. Plusieurs modèles du VCI ont été produits lors de ces précédentes recherches et ils ont permis d'évaluer les mécanismes de vérification statique et dynamique du VCI. Cette évaluation n'a pour le moment pas abordé la vérification de programmes non-linéaires, c'est-à-dire des programmes comportant des instructions de sauts.

La littérature dans le domaine de la vérification formelle est très vaste. De nombreuses méthodes formelles ont ainsi été utilisées pour faire de la génération de tests sur des systèmes critiques comme Java Card. L'utilisation de la méthode VTG pour la vérification de systèmes est une pratique encore récente. Elle n'a pour le moment été appliquée qu'avec la méthode formelle Event-B.

Méthodologie

Notre travail vise, dans un premier temps, à améliorer le modèle du VCI proposé par Savary et al. en y intégrant une modélisation des instructions réalisant des sauts simples (ex : goto) dans un programme Java Card. On souhaite, par cette amélioration, pouvoir étudier les mécanismes qui interviennent au sein du VCI lors de la vérification de programmes non-linéaires, c'est-à-dire des programmes comportant des instructions de saut.

Dans un second temps, notre travail consiste à utiliser la méthode B de manière à contribuer à l'avancée des outils et des méthodes qui constituent le VTG.

Lors de chaque expérimentation, un modèle formel fonctionnel est produit et vérifié par de l'exploration de modèle (*model-checking* en anglais). On extrait de ce travail de modélisation une suite de tests fonctionnels abstraits que l'on concrétise à l'aide d'outils de concrétisation. Les tests fonctionnels concrets sont ensuite exécutés sur le VCI afin d'étudier son comportement. Si le comportement observé est conforme à la spécification, nous réalisons une mutation de ce modèle pour en extraire des tests de vulnérabilité abstraits. La génération des mutations est réalisée manuellement ou automatiquement, suivant l'applicabilité des outils du VTG à l'expérimentation.

Nous établissons différentes mesures, comme la couverture du test généré, le nombre de tests générés et le temps mis pour générer ces tests, de manière à pouvoir comparer les différentes expérimentations entre elles.

Résultats

Les différentes réussites et échecs rencontrés lors de nos expérimentations nous ont permis de formaliser de manière détaillée la méthodologie utilisée par Savary et al. dans leurs expérimentations et l'adapter au contexte non-linéraire. Nous avons également pu adapter la chaîne d'outils du VTG à ce nouveau cas d'étude pour générer de manière automatique des tests de vulnérabilité applicables au VCI.

Notre travail de modélisation du VCI a abouti à trois modèles. Deux de ces modèles utilisent la méthode Event-B. Le dernier modèle est écrit en B classique. Nous avons ainsi pu démontrer qu'il est possible d'étendre la chaîne d'outils du VTG aux modèles formels en B classique.

Structure du mémoire

Dans le premier chapitre nous énonçons les notions nécessaires à la compréhension des thèmes abordés dans ce mémoire. Nous présentons ainsi dans une première section les notions relatives aux tests de vulnérabilité, puis nous introduisons les méthodes formelles pour permettre une meilleure compréhension de la méthode VTG.

Nous présentons ensuite l'état des connaissances actuelles en matière de génération de tests de vulnérabilités appliqués à Java Card.

Dans le troisième chapitre nous décrivons la méthode de génération de test proposée pour atteindre nos objectifs.

Dans le quatrième chapitre, nous appliquons cette méthodologie à nos cas d'étude et présentons les résultats de nos recherches.

Enfin, dans le cinquième chapitre, nous présentons les différents outils qui ont été développés lors de nos expérimentations.

Chapitre 1

Fondements

Les notions énoncées dans ce chapitre sont essentielles à la bonne compréhension du travail présenté. Après une courte introduction au test de logiciel, nous opposons l'approche de test classique à l'approche à base de modèle. Les méthodes formelles, dont découlent les techniques de test à base de modèle, sont présentées plus en détail dans une seconde section. Les cartes à puces Java Card, ainsi que la structure de leurs programmes, sont présentées dans une troisième section.

1.1 Introduction au test de logiciel

> *« Le test de logiciel est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou pour identifier les différences entre les résultats attendus et les résultats obtenus »* - IEEE (Standard Glossary of Software Engineering Terminology)

Il existe différentes méthodes pour tester un logiciel. Déterminer la méthode de tests adéquate pour un logiciel donné est une tâche complexe. Il convient, entre autres, de connaître la largeur du périmètre à étudier et des ressources, en temps et en main d'oeuvre, dont on dispose.

La connaissance de l'implémentation du SUT peut varier selon les cas. On parlera ainsi de test en boîte blanche (*white-box testing* en anglais) ou de test en boite noire

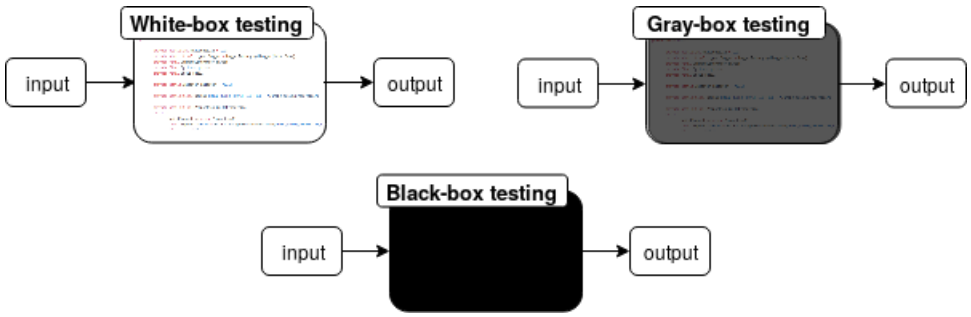

FIGURE 1.1 – Mise à disposition de l'implémentation du SUT selon le type de test. Dans le cas du *white-box testing*, l'accès au code source est complet.

(*black-box testing* en anglais) suivant si l'on connaît, ou non, l'implémentation du SUT (voir figure 1.1).

L'ensemble des valeurs à tester est bien souvent infini. Un des objectifs du test de logiciel est donc de sélectionner les cas de tests les plus susceptibles de mettre le système en défaut [11].

La sélection exhaustive utilise toutes les valeurs du domaine de définition pour constituer les cas de tests.

La sélection combinatoire part du principe que la mise en défaut d'un système peut provenir de la combinaison de plusieurs facteurs. Elle peut ainsi combiner dans un même ensemble deux espaces d'états, ou parlera alors de combinaison *pairwise*, ou bien plusieurs espaces. On parlera alors de combinaison *n-wise*.

La sélection aux bornes ou test par classe d'équivalence, consiste à tester si les valeurs en limite du domaine de définition sont acceptées ou non par le SUT. Ces tests sont relativement simples à mettre en place mais ils offrent une couverture rarement suffisante. En effet, ces tests se limitent aux bornes de ce qui est accepté par le SUT. On les combine donc souvent avec des tests aléatoires.

La sélection aléatoire consiste à prendre des valeurs au hasard pour constituer les cas de tests. Ils permettent d'obtenir un échantillonnage uniforme des domaines de définition. Ce type de tests est un bon complément des tests aux limites. Il est facilement automatisable, cependant, il est parfois difficile de garantir l'indépendance des variables et donc de s'assurer de la validité du test. Utiliser un oracle, en complément de cette technique de test, permet d'obtenir un bon

niveau de confiance.

La mise en pratique de ces différentes méthodes se divise en deux approches que nous opposons dans les deux sous-sections suivantes.

1.1.1 L'approche classique

L'approche traditionnellement utilisée en test de logiciel se compose de trois étapes.

L'élaboration des cas de tests consiste à définir le scénario de test, son contexte d'exécution et les critères de réussite ou d'échec à partir des spécifications du SUT.

L'exécution des tests et l'interprétation des résultats est l'étape lors de laquelle les cas de test sont exécutés sur le SUT. Dans le cas où la réponse du SUT diffère de la réponse attendue, on cherche à déterminer la raison de cet échec.

La vérification des caractéristiques du test permet d'évaluer la qualité du jeu de tests et du processus de test. On mesure la couverture des tests par rapport aux exigences fixées lors de la première phase.

Plusieurs méthodes de tests sont dites «classiques». La méthode manuelle en est la plus ancienne et est encore largement utilisée dans l'industrie. Elle consiste à rédiger dans un premier temps un plan de test, dans lequel sont définis les objectifs, la stratégie, la fréquence et la quantité de tests à générer. Les cas de tests à réaliser sont ensuite décrits et rassemblés dans un document qui est remis à la personne en charge de leur exécution. Chaque test est alors exécuté manuellement puis on compare les résultats attendus avec les résultats obtenus.

Cette méthode de génération nécessite d'avoir une bonne connaissance du SUT pour être efficace. Elle peut donc se révéler coûteuse suivant la taille du périmètre à évaluer.

La méthode par script permet d'automatiser l'étape d'élaboration et d'exécution des tests. Elle nécessite de disposer d'un point de contrôle et d'observation pour transmettre les valeurs au SUT et en observer les réactions. Le *fuzzing*, ou test à données aléatoires, fait partie de cette classe de tests. Il est souvent utilisé pour faciliter la recherche de vulnérabilités dans de gros projets [4, 13]. L'envoi de données potentiellement mal formées au SUT peut se faire de manière plus ou moins « intelligente » en fonction de la connaissance du système étudié et du type de fuzzer utilisé [16]. Ce procédé a l'avantage de pouvoir générer un très grand nombre de tests avec peu d'intervention humaine. Cependant, il n'est pas possible de générer avec cette méthode des tests *a priori* pertinents pour le système sous test. On génère par conséquent un grand nombre de faux positifs.

Parmi les autres méthodes de tests traditionnelles, on peut également citer la méthode par «*capture/replay*» qui consiste à capturer les interactions avec le SUT lors de l'exécution pour les répéter lors d'une exécution future. Il est également possible d'abstraire des tests à l'aide de mots clés puis de les remplacer par des fragments de code. Le haut niveau d'abstraction de cette méthode dite «*keyword-driven*» permet à des non-développeurs de lire et écrire facilement des cas de tests. Cependant, les fragments de code à intégrer nécessitent encore une intervention humaine. L'approche à base de modèle permet de répondre à ce problème.

1.1.2 L'approche à base de modèle

Le test à base de modèles, ou *model-based testing* en anglais (MBT), permet d'automatiser l'étape d'élaboration des cas de tests. Au lieu de rédiger manuellement des cas de test, l'effort est concentré sur l'écriture d'une spécification formelle du SUT[17]. Une fois le modèle obtenu, il est possible d'en extraire une grande quantité de tests de manière automatique en utilisant un explorateur de modèle comme ProB.

ProB, sur lequel nous reviendrons dans la suite de ce chapitre, utilise deux principaux algorithmes pour générer des tests.

Le model-checking based testing (MCM) est un algorithme qui consiste à explorer l'ensemble de l'espace d'états d'un modèle jusqu'à ce qu'un critère de cou-

verture soit satisfait. Cet espace d'état contient l'ensemble des initialisations possibles pour une machine ainsi que les différentes valeurs des constantes.

Le constraint-based testing (CBC) ne se base pas sur un ensemble d'état complet mais s'efforce plutôt d'explorer l'ensemble des chemins accessibles (*"feasible"*). À la différence du MCM, le CBC instancie les constantes du modèle et l'ensemble des paramètres de chaque opération. Il utilise ensuite un algorithme de parcours d'arbre en largeur ("breadth-first search") pour extraire les chemins satisfaisant un critère de couverture.

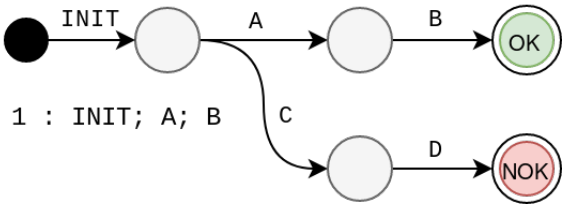

FIGURE 1.2 – L'exploration de ce modèle nous permet d'obtenir trace valide.

La figure 1.2 est un exemple de génération de tests fonctionnel par MBT. Le graphe représente l'ensemble des chemins qu'il est possible d'explorer en animant un modèle donné. On a ici contraint l'explorateur de modèle à ne conserver que les traces qui aboutissent dans un état "OK". L'explorateur n'a donc conservé que la trace *INIT ; A ; B*.

La génération de test par MBT peut générer un nombre très important de tests. Elle nécessite donc de déterminer au préalable des critères de sélection pour obtenir une suite de test pertinente pour le SUT. L'utilisation de cette méthode de test permet d'obtenir une bonne couverture, et donc un fort niveau de confiance dans le SUT, à condition que la spécification fournie soit complète et précise. Elle nécessite un plus long temps de conception et c'est pour cette raison qu'elle n'est souvent utilisée que pour l'évaluation des systèmes critiques, dont Java Card fait partie.

1.1.3 Le test par mutation

Le terme "test par mutation" désigne traditionnellement le test par **mutation de programme**. Cette méthode vise à évaluer la qualité d'un jeu de test en réalisant des mutations sur l'implémentation du système sous test. En observant la capacité du jeu de tests à détecter ces mutations, on peut déterminer sa fiabilité et ainsi développer des suites de tests plus efficaces. Cette technique a été appliquée à différents langages de programmation, dont Fortran [10], C [1], ADA [14] et Java [9]. L'inconvénient de cette méthode c'est qu'elle ne vérifie pas que les mutations introduites dans l'implémentation du SUT modifient réellement le résultat final. Il est ainsi possible d'obtenir des tests structurellement différents mais sémantiquement équivalents.

Il est possible d'appliquer les mêmes principes à des spécifications formelles. On parlera alors de **mutation de spécification**. Cette méthode consiste à appliquer des règles de négation à certaines formules de la spécification formelle d'un SUT. Le modèle transformé obtenu est alors appelé modèle "mutant". Contrairement à la mutation de programme, il est possible de définir des propriétés qui seront vérifiées sur le modèle originel, sur les modèles mutants et sur le processus de mutation. Il est cependant possible d'utiliser le processus de mutation de programme en complément de la mutation de spécification.

Dans [2], Savary énonce les règles de négations suivantes, où $p_1, ..., p_n$ sont des prédicats, $b_1, ..., b_n$ sont des variables booléennes, $n_1, ..., n_n$ sont des variables de types entier naturel, et neg est une fonction représentant l'application des règles à un prédicat. Ces règles sont appliquées de façon récursive par un parcours de l'arbre syntaxique du prédicat.

- $neg(p_1 \wedge p_2) \rightarrow \{neg(p_1) \wedge p_2, p_1 \wedge neg(p_2), neg(p_1) \wedge neg(p_2)\}$
- $neg(p_1 \vee p_2) \rightarrow \{neg(p_1) \wedge neg(p_2)\}$

- $neg(n_1 = n_2) \rightarrow \{n_1 < n_2, n_1 > n_2\}$
- $neg(n_1 > n_2) \rightarrow \{n_1 = n_2, n_1 < n_2\}$
- $neg(n_1 < n_2) \rightarrow \{n_1 = n_2, n_1 > n_2\}$
- $neg(n_1 \geq n_2) \rightarrow \{n_1 < n_2\}$

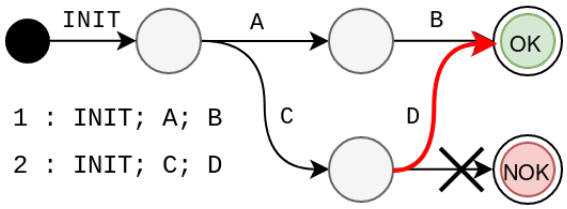

FIGURE 1.3 – Une fois le modèle mutant obtenu, on peut utiliser un explorateur de modèle pour extraire des tests abstraits. À la différence du modèle présenté en figure 1.2, les traces extraites pourront ici représenter des fautes.

— $neg(n_1 \leq n_2) \rightarrow \{n_1 > n_2\}$
— $neg(n_1 \neq n_2) \rightarrow \{n_1 = n_2\}$

— $neg(b_1 = b_2) \rightarrow \{\neg(b_1 = b_2)\}$
— $neg(b_1 \neq b_2) \rightarrow \{b_1 = b_2\}$

Cette méthode est efficace pour générer des modèles mutants à partir de modèles fonctionnels mais le coût en temps pour réaliser le modèle initial et en dériver des modèles mutants est important.

1.1.4 La méthode Vulnerability Test Generation

Pour palier le problème du coût de la génération des modèles mutants, Savary *et al.* proposent dans [15] la méthode VTG («Vulnerability Test Generation», génération de tests de vulnérabilité). Elle décrit un ensemble de procédures à suivre pour générer automatiquement des tests de vulnérabilités concrets à partir d'un modèle formel fonctionnel, comme illustré sur la figure 1.4. Nous présentons les deux premiers composant de cette illustration dans les sous-parties suivantes. Le dernier composant est propre au SUT et est détaillé en section 1.3.2.

Génération automatique des modèles mutants

Le modèle de Savary et al. utilise le langage formel Event-B. Ce langage, que nous présentons plus en détail en partie 1.2.3, permet de concevoir formellement la spécification d'un programme sous forme de contextes et de machines abstraites. Une

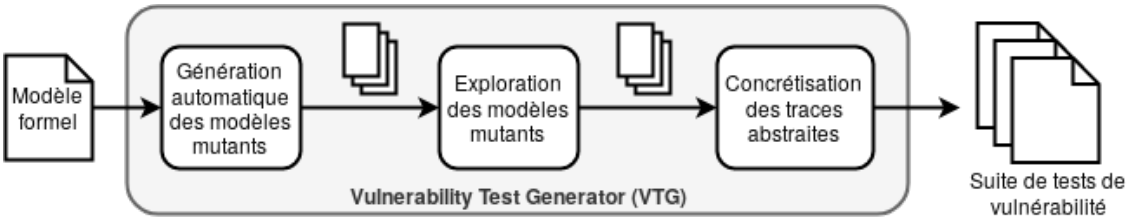

FIGURE 1.4 – Processus de génération de tests de vulnérabilité à l'aide de la méthode VTG et de ses outils associés

machine est constituée d'événements dont l'exécution est conditionnée par la validité d'une précondition appelée «garde». La post-condition d'un événement est appelée «action».

Pour générer les modèles mutants, ils proposent dans [2] l'algorithme 1. Cet algorithme en trois parties prend en entrée un modèle Event-B et produit en sortie un ensemble de modèles Event-B mutés.

Dans une première partie, chacune des gardes des événements du modèle sont remplacées par deux gardes *grd* et *grd_t* représentant respectivement la conjonction des gardes que l'on souhaite nier, préalablement suffixées par *_t*, et la conjonction des gardes que l'on souhaite conserver, qui ne sont pas suffixées.

Dans une seconde partie, l'algorithme utilise la fonction *neg*, telle que définie en section 1.1.3, pour muter la garde *grd_t*. Chaque mutation obtenue est placée dans un ensemble d'événements *RW* («*rewrite*»).

Enfin, il génère l'ensemble de modèles mutés en intégrant chaque événement *RW* dans un modèle mutant distinct. Pour repérer les événements ayant subi une mutation, il ajoute le suffixe *_mut* suivi d'un numéro d'identification unique.

Exploration des modèles mutants

L'exploration des modèles mutants permet d'extraire des tests de vulnérabilité sous la forme de traces abstraites. Savary et al. proposent deux approches pour réaliser cette extraction : l'une utilisant de la vérification de formules de logique temporelle linéaire (LTL), et la seconde basée sur de la résolution de contrainte. Nous avons choisi de nous intéresser à la seconde.

Algorithm 1 Extraction de modèles mutants

Input : m : Event-B model
Output : M ' : set of Event-B models

for each event e of m **do**
    rewrite the guard of e into two guards :
        grd, the conjunction of all guards of e without suffix **_t* ;
        grd_t, the conjunction of all guards of e with suffix **_t* ;
for each event e of m **do**
    $e.RW := neg(grd\_t)$;
for each event e of m **do**
    for each rw in $e.RW$ **do**
        add a new model m' to M' such that
            $m' := m$;
            $m!.events := m'.events \cup e'$, where e' is defined as follows :
                $e' := e$;
                replace $e'.grd\_t$ by rw ;
                add guard "$mut = FALSE$" to e ;
                add action "$mut := TRUE$" to e' ;

La sélection de tests par résolution de contrainte, ou «*constraint based criteria*» (CBC) permet de définir et d'imposer, pour chaque trace extraite du modèle mutant, la résolution d'une contrainte. Dans l'algorithme du VTG, chaque modèle muté contient une variable booléenne *fault*. Pour chaque trace extraite du modèle mutant, on impose $fault = TRUE$. Savary et al. limitent également la longueur maximale de trace pour chaque extraction. La génération des traces se fait dans un ordre croissant de longueur.

1.2 Les méthodes formelles

Les méthodes formelles permettent de raisonner rigoureusement sur des systèmes informatiques, afin de démontrer leur validité par rapport à une spécification [7]. Il existe de nombreuses méthodes formelles [8]. Nous nous intéressons aux méthodes dites à base de modèle, comme Z [18], B et Event-B. Nous présenterons ces deux dernières dans cette section. La sous-section qui suit vise à fournir les outils mathématiques nécessaires à la compréhension de ces deux méthodes.

1.2.1 Introduction à la logique du premier ordre

Les méthodes formelles à base de modèles utilisent la logique du premier ordre. Elles permettent d'exprimer des faits de manière formelle, c'est-à-dire sans ambiguïté. Il est ensuite possible de raisonner par rapport à ces faits et en déduire de nouveaux faits.

Une formule de logique du premier ordre se construit à partir de variables reliées entre-elles par des prédicats et des connecteurs logiques. Un prédicat $r(x_1, \ldots, x_n)$ peut être vu comme une fonction booléenne ou comme une relation. Les exemples typiques en arithmétique sont les opérateurs de comparaison comme $=$, $<$, etc. Les connecteurs logiques $\neg, \wedge, \vee, \oplus, \implies$ et $\iff$ permettent d'exprimer respectivement la négation, la conjonction, la disjonction, la disjonction exclusive, l'implication et l'équivalence. La valeur d'une formule, ou valeur de vérité, est déterminée à partir des tables de vérité propres au connecteur logique utilisé et en associant aux prédicats une relation ou une fonction booléenne. Les variables d'une formule peuvent être quantifiées existentiellement ($\exists$) ou universellement ($\forall$).

1.2.2 La méthode B

La méthode B est une méthode formelle permettant de raisonner sur des systèmes complexes. Le langage B est basé sur la théorie des ensembles, la logique du premier ordre et le langage des substitutions généralisées. Il permet de concevoir formellement la spécification d'un programme sous forme de machine abstraite. Cette machine abstraite peut ensuite être concrétisée en réalisant ce qu'on appelle un raffinement.

Pour présenter la structure d'un modèle B, nous prenons l'exemple d'un programme permettant la modification de la valeur d'une variable *val* par incrémentation. Tout au long de l'exécution du programme, cette valeur *val* doit rester dans un intervalle dont la borne supérieure est une constante *MaxVal* telle que $MaxVal \in \{1, ..., 8\}$. Le programme B modélisant cette spécification est le suivant :

MACHINE Incrementeur
CONSTANTS

```
MaxVal
```

PROPERTIES

$\texttt{MaxVal} \in 1 \mathinner{..} 8$

VARIABLES

$\texttt{val}$

INVARIANT

$\texttt{val} \in 0 \mathinner{..} \texttt{MaxVal}$

INITIALISATION

$\texttt{val} := 0$

OPERATIONS

Inc =
PRE **val** $<$ **MaxVal** *THEN*
val := val + 1
END

Les spécifications en B classique utilisent plusieurs composants. Le composant *Machine* représente le modèle global. Il est composé de *Constants* qui décrivent les constantes du modèle, dont les propriétés sont définies en partie *Properties*. Un *Invariant* est une condition sur une variable définie dans la clause *Variables*. La violation d'un invariant lors de l'exploration du modèle entraîne une erreur.

1.2.3 La méthode Event-B

La méthode Event-B est une évolution du B. Elle remplace la notion d'opérations par une notion d'événements. Cette vision plus générale lui permet de s'appliquer aisément à des domaines plus variés que la vérification de programmes.

De plus, la notion de raffinement en Event-B est un peu différente de celle en B. En B, il n'est pas possible d'ajouter de nouvelles opérations lors du raffinement. Chaque opération est réécrite pour prendre en compte des variables concrètes. Les préconditions des opérations sont ainsi élargies pour s'adapter aux ensembles de définition des variables concrètes. On utilise un invariant de collage pour s'assurer du bon raffinement des données.

En Event-B, il est possible d'ajouter des opérations lors de la création d'un nouveau raffinement. Le rôle du raffinement est alors de renforcer petit à petit les gardes pour construire, couche après couche, une spécification complète. Un modèle Event-B se compose de deux parties : les contextes et les machines.

Contexte

Les contextes Event-B permettent de modéliser les constantes et les axiomes, qui constituent la partie statique du modèle. Il est possible de raffiner un contexte pour le représenter à différents niveaux d'abstraction. Le contexte Event-B du programme présenté en section 1.2.2 est le suivant :

CONTEXT C0
CONSTANTS

`MaxVal`

AXIOMS

`axm1 : MaxVal ∈ 1 .. 8`

END

La constante que l'on souhaite soumettre à une ou plusieurs contraintes est déclarée dans la clause *CONSTANTS*. Ces contraintes sont définies dans la clause *AXIOMS*. Les axiomes sont des prédicats du premier ordre qui doivent être vrais pour pouvoir instancier un contexte. Ici l'axiome *axm1* contraint la valeur de *MaxVal* dans l'intervalle $[1; 8]$.

Machine

Les machines Event-B représentent la partie dynamique du modèle. Une machine est constitué de différents événements. La garde est un prédicat qui, selon s'il est vrai ou faux, autorise ou interdit l'exécution d'un événement. Si un événement se produit, l'action met à jour la valeur de certaines variables. Les gardes et actions d'un événement constituent respectivement sa précondition et sa post-condition.

On obtient la machine suivante pour le cas du programme d'incrémentation présenté en partie 1.2.2 :

MACHINE M0
SEES C0
VARIABLES

`val`

INVARIANTS

`inv1 :` $\texttt{val} \in 0 \mathinner{..} \texttt{MaxVal}$

EVENTS
INITIALISATION

begin

`act1 :` $\texttt{val} := 0$

end

Event *evtInc* $\widehat{=}$

any

`post_val`

where

`grd1 :` $\texttt{post\_val} = \texttt{val} + 1$

`grd2 :` $\texttt{post\_val} \leq \texttt{MaxVal}$

then

`act1 :` $\texttt{val} := \texttt{post\_val}$

end

END

Cette machine est mise en lien avec le contexte *C0* défini précédemment par l'instruction *SEES C0*. La valeur de *MaxVal* utilisée dans cette machine est donc soumise aux contraintes définies dans le contexte *C0*. La variable à incrémenter est située dans la section *VARIABLES*. Elle est également soumise à certaines contraintes définies dans la section *INVARIANTS*.

Le premier événement de cette machine est l'événement *INITIALISATION*. Cet événement ne comporte pas de pré/post-conditions. Il est donc structuré sous la forme

BEGIN ... END. Il comporte une action *act1* dans la section *BEGIN* qui affecte la valeur 0 à la variable *val*.

Le second événement est l'événement *evtInc* qui incrémente la valeur de *val* de 1. Cet événement comporte des pré/post-conditions. Il possède donc une structure *ANY ... WHERE ... THEN ... END*. La variable locale *post_val* est déclarée dans la clause *ANY*, qui représente le choix non-déterministe d'une valeur pour les variables locales. La valeur de la variable locale *post_val* est soumise aux gardes définies dans la clause *WHERE*. Dans cet événement la garde *grd1* s'assure que la valeur d'arrivée sera égale à la valeur de départ incrémentée de 1. La garde *grd2* garantie que la valeur d'arrivée ne dépasse pas la constante *MaxVal*. Le fonctionnement des gardes est comparable à celui des paramètres de la clause *OPERATION* en B classique. L'action *act1* du bloc *THEN* affecte à *val* la valeur de *post_val*.

1.2.4 ProB et l'exploration de modèle

Les méthodes décrites précédemment ne permettent pas de prouver des propriétés sur les traces car il faudrait pour cela considérer le modèle de façon dynamique. On utilise donc des vérificateurs de modèles comme ProB. ProB permet l'animation, la résolution de contraintes et l'exploration de modèles dans de nombreux langages formels comme le B classique et Event-B. Il peut être utilisé pour trouver des impasses (*deadlock* en anglais) ou encore générer des cas de tests. L'exploration de modèle consiste à construire puis à explorer un graphe de transitions entre les différents états du système modélisé. Chaque étape d'exploration du modèle, c'est-à-dire chaque action effectuée sur le modèle, est consignée sous forme de «trace» de longueur finie.

En animant le modèle Event-B d'incrémenteur présenté dans la sous-section 1.2.3 on peut obtenir la trace suivante :

SETUP_CONTEXT(2) Définit le contexte. La définition du contexte consiste ici à affecter à $MaxVal$ une valeur située dans son intervalle définit dans le contexte $C0$. Dans le cas d'un $SETUP\_CONTEXT(2)$, cette valeur vaut 2.

INITIALISATION Initialise la variable *val*. Cette valeur est fixée à 0 dans la machine $M0$.

evtInc(1) Incrémente de 1 la valeur courante de *val*. On se retrouve donc dans un

état où *val* vaut 1. Dans cet état les gardes permettent à l'événement evtInc de survenir à nouveau.

evtInc(2) Incrémente de 1 la valeur courante de *val*. On se retrouve donc dans un état où *val* vaut 2. La seconde garde *grd2* : $post\_val \leq MaxVal$ sur l'événement evtInc ne permet pas de réitérer cet événement comme la borne supérieure est atteinte. Le système se retrouve donc en *deadlock*, c'est-à-dire un état où aucun autre événement ne peut survenir.

Une trace similaire peut être générée à partir du modèle en B classique. L'ensemble des états explorables sous ProB lorsqu'on initialise ce modèle avec $MaxVal = 1$, $MaxVal = 2$ et $MaxVal = 3$ est représenté par la figure 1.5.

1.2.5 ProB Java API

L'animateur et model-checker ProB met à disposition une API (interface de programmation applicative ou «*Application Programming Interface*» en anglais) permettant aux développeurs de charger et d'animer des modèles B, Event-B, CSB et Z. L'interface utilisateur de ProB, dans sa version 2.0, est elle-même basée sur cette API[1]. Cette API est écrite en Java et Groovy.

```
System.out.println("Load classical B Machine");
Path p = Paths.get(getClass().getResource("example.bcm").toURI());
StateSpace s = api.eventb_load(p.toAbsolutePath().toString());

System.out.println("Show random trace");
Trace t = new Trace(s);
for (int i = 0; i < 10; i++) {
    t = t.anyEvent(null);
}
System.out.println(t);
```

Programme 1.1 – Exemple de code Java utilisant l'API ProB 2.0

L'exemple 1.1 est un exemple d'animation d'un modèle Event-B par l'API Java ProB. La fonction `get` de la classe `Paths` permet de charger en mémoire un fichier. On l'utilise ici pour charger le fichier .bcm contenant la machine d'un modèle Event-B

1. https ://www3.hhu.de/stups/handbook/prob2/prob_developer.html

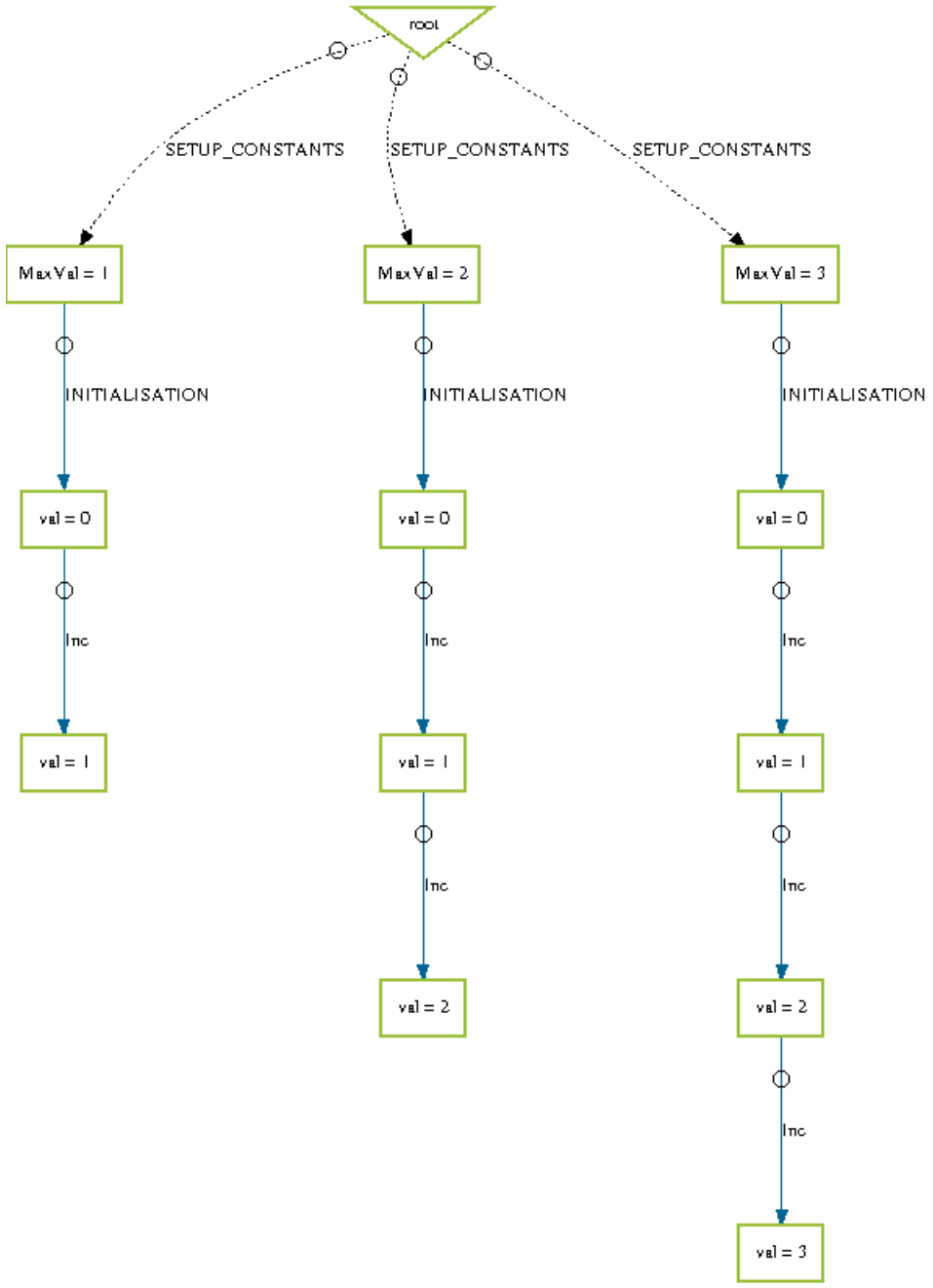

FIGURE 1.5 – Visualisation sous GraphViz de l'ensemble des états explorables par ProB

dans une variable `p`. On appelle ensuite la fonction `eventb_load` sur cette variable `p`. On démarre une nouvelle trace en créant une instance `t` de la classe `Trace` puis on exécute aléatoirement une série de dix événements en appelant la fonction `anyEvent`. Cette fonction exécute n'importe quel événement accessible sur la machine. Si aucun événement n'est accessible (deadlock) alors elle n'effectue aucune transition. Enfin, l'appel de la fonction `println` sur la variable `t` affiche les opérations accessibles, le nombre de transitions exécutées et le nom de la dernière transition effectuée.

1.3 Les cartes à puce Java Card

Qu'elles soient avec ou sans contact, bancaires, d'assurance maladie, étudiantes ou encore SIM, les cartes à puces sont des objets bien implantés dans notre quotidien. On distingue deux familles de cartes à puce : synchrones et asynchrones. Les cartes asynchrones, ou cartes à microcontrôleur, ont la particularité de posséder des mémoires vives et mortes ainsi qu'une interface d'entrée/sortie. Ces caractéristiques leur confèrent l'infinité de possibilités d'applications que nous connaissons aujourd'hui mais les rendent également plus vulnérables.

1.3.1 Notions générales

Les cartes à puce Java Card sont des «Smart Card» asynchrones. Elles utilisent le langage Java Card, un sous-ensemble du langage Java, conçu pour s'adapter aux contraintes en taille mémoire et en puissance de calcul imposées par l'architecture des cartes à puce. Elles permettent le chargement d'applications appelées Applet qui sont stockées dans la mémoire de la carte.

Notre recherche porte sur l'étude des instructions de branchement dans la JCVM (Java Card Virtual Machine) dans sa version 3.0.1 Classic Edition. En Java les structures utilisant des instructions de branchement sont les suivantes :

— while
— if, else if, else
— for

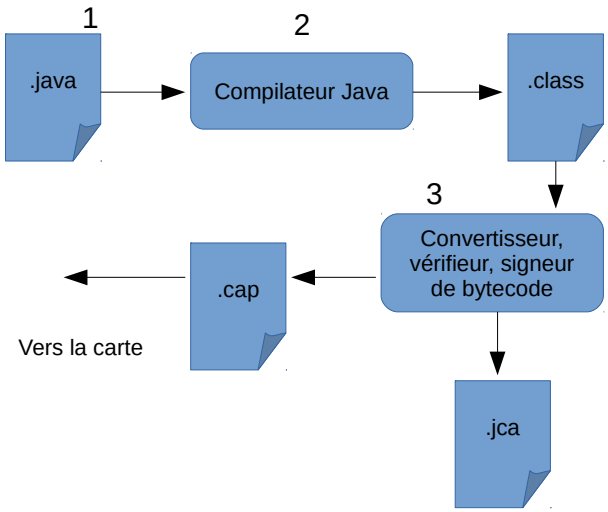

FIGURE 1.6 – Cycle de vie d'une applet Java Card

— switch
— try, catch, finally

Dans le cadre de cette expérimentation nous n'étudions que les branchements simples, c'est-à-dire des programmes dans lesquels les sauts ne vont que dans une seule direction (vers l'avant, vers l'arrière ou vers soit-même). Les instructions de la JCVM permettant d'effectuer ce type de branchement sont les suivantes :

— goto
— goto_w

La figure 1.6 décrit le cycle de vie d'une applet Java Card. Ce cycle de vie est composé d'une étape de compilation d'un programme Java puis d'une conversion permettant d'aboutir à un fichier .cap exécutable depuis une carte à puce Java Card et d'un fichier .jca textuel, écrit en langage Java Card Assembly.

1.3.2 Java Card Assembly

Comme vu dans la sous-section 1.3.1, l'étape de conversion génère un fichier binaire .cap et un fichier textuel équivalent .jca (Java Card Assembly). Le fichier JCA est

l'exacte représentation textuelle du format binaire CAP. Il nous est alors possible d'en modifier une portion puis d'utiliser l'outil de conversion *capgen* de Java Card pour générer un nouveau fichier CAP.

Le code 1.2 affecte à une variable la valeur 0 puis initie une boucle infinie incrémentant cette variable. Il a pour équivalence JCA le code 1.3.

Programme 1.2 – Code Java

```
short variable = 0;
while(true) {
        variable++;
}
```

Programme 1.3 – Code JCA

```
L0:     sconst_0;
        sstore_0;
L1:     sinc 0 1;
        goto L1;
```

En CAP les instructions ainsi que leurs arguments sont stockés dans un tableau unidimensionnel appelé «*byte array*». Lors de l'appel d'une instruction de saut, on spécifie en paramètre le décalage à réaliser par rapport à l'emplacement courant. Ce décalage peut être négatif (cas des branchements en arrière), positif (cas des branchements en avant) ou nul (cas des branchements vers soit-même).

En JCA, la représentation des sauts est un peu différente. Chaque appel vers une instruction de saut crée une étiquette Lx, dont x varie entre 0 et l'infini. On place ensuite l'étiquette crée juste avant l'appel de l'instruction vers laquelle créer le saut.

1.3.3 Le vérifieur de code intermédiaire

Le VCI représente la partie statique de la vérification des applications Java Card. Une application est dite valide si elle respecte la spécification Java Card, sinon elle est dite invalide. Une application acceptée est une application considérée comme valide par une implémentation du VCI, sinon elle est dite rejetée. Le rejet par le VCI d'une application valide est un comportement normal et ne représente pas une défaillance du VCI. En effet, la spécification définit un mécanisme de vérification qui est une approximation sûre et autorise à rejeter plus d'applications si nécessaire (dans le cas de contre-mesure par exemple). En revanche, l'acceptation d'un programme invalide est problématique. Ce comportement correspond à une défaillance susceptible de conduire à des failles de sécurité. La génération de tests de vulnérabilité sur le VCI consiste à trouver des cas correspondant à ce dernier comportement.

La vérification du byte code Java Card, ou code intermédiaire, a lieu en dehors de la carte dans le cas d'une vérification «*Off-Card*», par opposition à la vérification «*On-Card*». Dans le cadre de cette recherche nous nous intéressons à la première, représentée à l'étape 3 de la figure 1.6.

Un fichier CAP est constitué de 12 composants interdépendants. Il existe des composants de classe et des composants de méthode. C'est à partir de ce dernier que débute la vérification.

Dans le cas d'une vérification «*Off-Card*», la vérification du fichier .cap par le VCI s'effectue avant son chargement sur la carte. Elle se déroule en deux étapes : la lecture des données et la vérification de ces données. La vérification des données porte sur la structure et sur les types.

La vérification de structure s'assure de la validité des champs concernant les bornes, les en-têtes et les «*exception handlers*». Si la vérification de structure échoue, le VCI rejette l'application sans effectuer la vérification de type. Si elle réussit, l'application est envoyée au vérifieur de type.

La vérification de type sert à garantir que le bytecode n'effectue pas de conversions illégales de type ou que les types utilisés sont appropriés aux opérations. Elle simule l'exécution de chaque instruction en n'utilisant que les types pour chaque variables. L'application est considérée comme acceptée par le VCI si le vérifieur de type la considère valide.

Dans [15], Savary et al. étudient le fonctionnement de la vérification de type pour les programmes linéaires et les programmes non-linéaire. Les programmes non-linéaires sont des programmes qui comportent des instructions de branchement. Les mécanismes de vérification de boucle que nous souhaitons évaluer n'interviennent que dans ce type de programme.

Fonctionnement de la vérification de type des programmes non-linéaires

La spécification Java définit un ensemble de types, dont la hiérarchie est représentée sous forme d'arbre en figure 1.7. L'élément Top est un élément fictif qui représente le sommet de l'arbre. Lors de la réexécution d'une instruction, si le VCI détecte la création d'une boucle alors il calcule le *supremum* entre le type calculé et le type de chacun des successeurs, c'est-à-dire le premier type commun dans la hiérarchie des

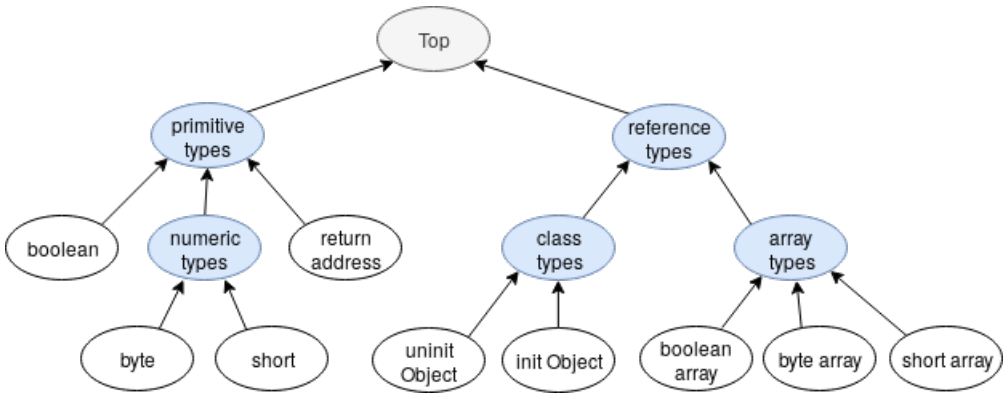

FIGURE 1.7 – Arbre des types en Java Card

types. Le VCI considère un programme comme invalide si, lors de la vérification, une de ses instructions est appelée avec un type incorrect par rapport à sa spécification ou bien si le supremum est l'élément *Top*. Si le calcul du supremum aboutit à l'élément *Top*, alors on dit que les deux types en entrée du calcul du supremum sont incompatibles.

Savary et al. proposent dans [15] le patron d'algorithme 2 représentant les opérations réalisées par le VCI lors de la vérification de type. Cet algorithme prend trois variables en entrée. Ces trois variables sont des tableaux dont l'indice est situé entre 1 et une constante *maxpc*, qui représente la valeur maximale du pointeur d'instruction (ou *pointer counter*) du programme à vérifier. La variable p contient la liste des instructions du programme. La variable fs contient la «*Frame State*», c'est-à-dire l'état de la pile d'opérandes et des variables locales pour chaque instruction du programme p. La variable b est un vecteur de booléens initialisé à *False*. Elle est utilisée dans la suite de l'algorithme pour marquer les instructions vérifiées par le VCI.

Pour chaque instruction du programme, l'algorithme vérifie d'abord si les préconditions de l'instruction sont vérifiées. Si c'est le cas, alors il regarde si les éléments de la *Frame* qui succède à la *Frame* courante sont compatibles avec les éléments en post-condition de l'instruction. Le booléen pour l'indice en cours est, le cas échéant, passé à *True*. La *Frame* qui succède subit alors une inférence de type. Ses types sont remplacés par le supremum entre le fs courant et le fs en post-condition selon l'arbre d'héritage des types Java Card (voir figure 1.7).

Algorithm 2 Patron de l'algorithme du VCI

input : $p : 1..maxpc \rightarrow JBC\_INSTRUCTION$
var : $fs : 1..maxpc \rightarrow FRAME\_STATE$
var : $b : 1..maxpc \rightarrow BOOLEAN$

1: **procedure** BYTECODEVERIFIER(p)
2: $fs := (\{1\} \times \{InitialStateFrame\}) \cup (2..maxpc \times \{\bot\})$
3: $b := (\{1\} \times \{true\}) \cup (2..maxpc \times \{false\})$
4: **while** $\exists i : i \in 1..maxpc \cdot b(i)$ **do**
5: pick some i such that b(i)
6: $b(i) := false$
7: **if** $p(i)$ satisfies the precondition defined in [12] **then**
8: $post :=$ frame state obtained by executing $p(i)$ on $fs(i)$
9: $S :=$ successor instructions of $p(i)$ in p
10: **if** $\forall s \in S \cdot$ is type compatible with $fs(s)$ **then**
11: $b := b \lhd (\lambda s \cdot s \in S \wedge post \neq fs(s) \mid true)$
12: $fs := fs \lhd (\lambda s \cdot s \in S \mid post \vee fs(s))$
13: **else**
14: **return** invalid program
15: **else**
16: **return** invalid program
17: **return** valid program

Cet algorithme est une base de travail mais ne constitue pas l'algorithme final pour notre cas d'étude. En effet, la vérification des types ne nécessite pas de connaître la valeur des variables locales. Nous ne conserverons donc que l'état de la pile d'opérandes.

1.3.4 Application de la méthode VTG à la vérification du VCI

L'application de la méthode VTG au vérifieur de code intermédiaire Java Card a permis à Savary et al. de générer 223 tests de vulnérabilité concrets en 45 minutes. Les tests produits sont des applets Java Card au format CAP, c'est-à-dire sous forme de code intermédiaire.

Les auteurs proposent de représenter le VCI en deux modèles. Un premier modèle Event-B représente les contraintes statiques de la vérification de structure. Un second modèle Event-B représente les contraintes dynamiques de la vérification de type. Ces deux modèles sont vérifiés formellement en utilisant des méthodes de preuves et d'exploration de modèle présentées précédemment.

Une fois les propriétés de la spécification du système modélisées, Savary et al. cherchent à obtenir un modèle représentant un comportement anormal du VCI. Pour cela ils utilisent des techniques de mutation de spécification. Ces techniques consistent en une transformation automatique du modèle formel.

Une fois le modèle muté obtenu, les auteurs peuvent générer des tests de vulnérabilité en appliquant un algorithme de résolution de contraintes. Les tests se présentent sous la forme de fichiers CAP valides.

Chapitre 2

État de l'art

Dans ce chapitre nous présentons l'état de l'art dans le domaine de la génération automatique de tests de vulnérabilités.

Il s'appuie sur quatre documents qui traitent du test à base de modèle, des tests combinatoires, des cartes à puce Java Card et du VCI.

2.1 La méthode VTG appliquée au VCI

La méthode « *Vulnerability Test Generation*», ou VTG [15], est le fruit de 8 années de recherches entreprises par Savary et al. Notre travail de recherche s'inscrit dans la continuité de ces travaux. La méthodologie VTG ainsi que le cas d'étude du VCI sont similaires.

Dans [2], l'auteur propose une nouvelle approche de test en boîte noire permettant de générer automatiquement des tests de vulnérabilité. Cette méthode, le VTG, est basée sur des tests à base de modèle et de la mutation de spécification. La mutation de spécification est une transformation de la spécification du SUT selon certaines propriétés. Un modèle Event-B est produit puis muté selon des règles de négation de manière à transformer les propriétés de la spécification du VCI. L'auteur est ainsi en mesure d'extraire des cas de tests de vulnérabilité. Ce document présente, dans un premier temps, la théorie des cartes à puces Java Card ainsi que les mécanismes de vérifications de type et de structure des fichiers CAP effectués par le VCI. Cette introduction permet à l'auteur de présenter, dans un second temps, les différentes attaques

auxquelles sont vulnérables les cartes à puces Java Card, et plus particulièrement les attaques dîtes "logiques". Ces attaques reposent sur des failles au niveau des logiciels et leur dangerosité est proportionnelle aux technologies auxquelles ont accès les attaquants. L'auteur explique qu'il est donc nécessaire de disposer de nouveaux outils pour prévoir, détecter et limiter l'impact de ces attaques.

Dans [3] le modèle du VCI est étendu et le processus de génération de modèles mutants est accéléré grâce à un programme dédié à la génération de tests de vulnérabilités qui se base sur ProB.

Notre méthodologie se base sur celle présentée dans [2]. Le cas d'étude auquel nous appliquons cette méthodologie est une extension du cas d'étude de Savary prenant en compte les instructions non-linéraires, c'est-à-dire les instruction capables de créer des boucles. Nous devons donc faire évoluer la chaîne d'outils du VTG de manière à traiter ces nouvelles instructions. L'API ProB utilisée lors de ces expérimentation a subi un changement de version. Lors de ce changement de version, plusieurs appels API utilisés par le VTG ont été supprimés ou renommés. L'outil n'est donc, en l'état, plus fonctionnel. Nous devons par conséquent réaliser les mises à jour nécessaires ou trouver des solutions palliatives.

2.2 Validation de l'implémentation du VCI par tests combinatoires

Dans [5], Calvagna et al. proposent une technique pour automatiser complètement la validation de l'implémentation du VCI.

Cette technique repose sur une modélisation formelle de la Java Card Virtual Machine. L'exploration du modèle permet d'obtenir toutes les séquences possibles de passages d'un état autorisé à un état non-autorisé. Ces séquences sont ensuite combinées les unes avec les autres au niveau des points de branchements (*while/if*) de manière à obtenir des suites de tests de vulnérabilité, comme visible sur la figure 2.1 Cette entremêlement des flots d'exécution leur permet d'évaluer les mécanismes d'inférence de type de la JCVM.

Le cas d'étude de cet article est similaire au nôtre puisqu'il traite de la vérification

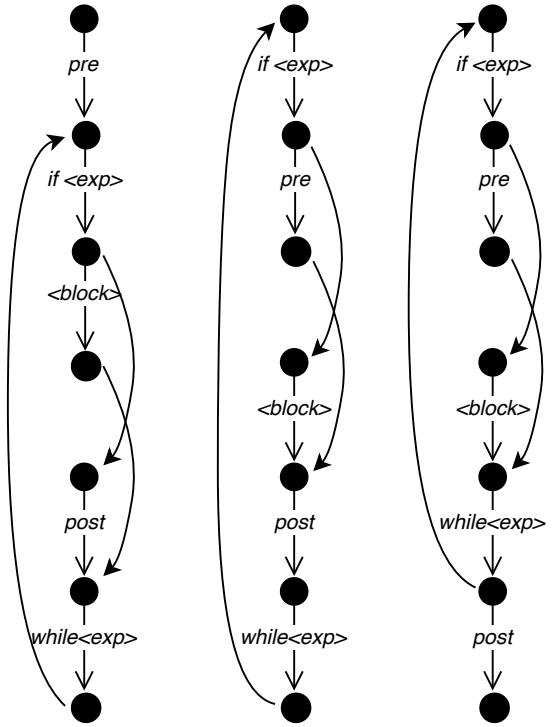

FIGURE 2.1 – Trois exemples de flots d'exécution entremêlés

du VCI. Les mécanismes de vérification du VCI analysés sont également ceux relatifs au contrôle de flot. Cependant, dans cette méthode les flots d'exécution sont entremêlés, c'est-à-dire qu'on injecte des instructions dans un flot existant. La méthode VTG utilisée dans nos recherches consiste à générer un nouveau fichier CAP à chaque test. Dans notre méthode le flot d'exécution est donc totalement nouveau à chaque test généré.

2.3 Taxonomie et classification des méthodes de test à base de modèle

Il existe différentes façons de modéliser la spécification d'un système, de générer des tests puis de les exécuter. Dans [6], Felderer et al. s'appuient sur 119 publications pour dresser un état de l'art et une classification des différentes méthodes de test

à base de modèle. Ce document nous permet de positionner la méthode VTG dans l'état de l'art des méthodes de test à base de modèle.

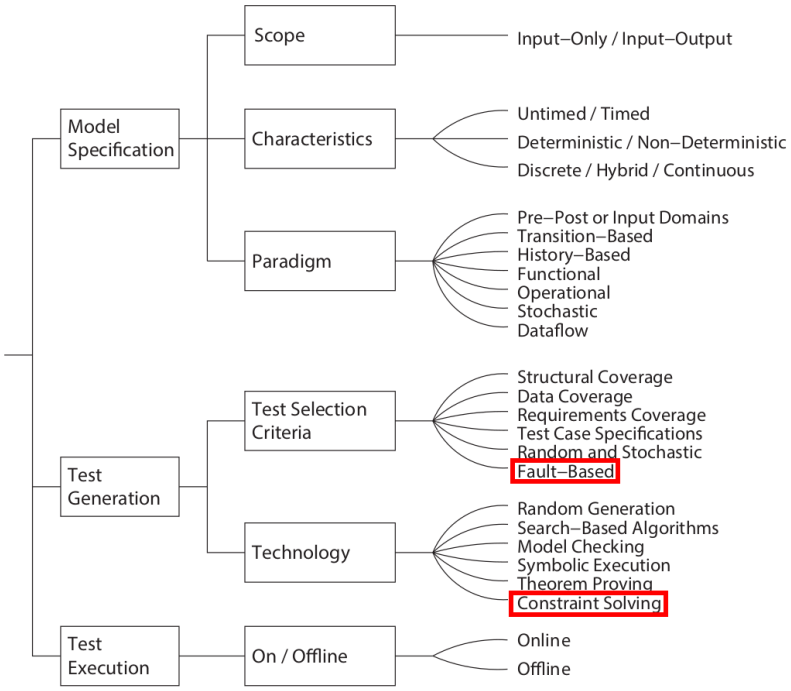

FIGURE 2.2 – Position de notre recherche dans la classification proposée par Felderer et al.

2.3.1 Classification des méthodes de spécification du modèle

L'auteur classe les méthodes de spécification du modèle selon trois critères : la portée des tests, les caractéristiques des tests et le paradigme de modélisation.

Dans le cas du VCI, la portée des tests s'étend des valeurs en entrée du SUT jusqu'aux réponses attendues en sortie du SUT. Un des caractéristique de notre modèle du VCI est qu'il modélise uniquement la pile d'opérande et non les valeurs des variables locales car elles ne sont pas nécessaires à notre cas d'étude. Le paradigme de modélisation est fonctionnel de manière à représenter le fonctionnement normal du système sous test. Le modèle fonctionnel est ensuite muté pour en extraire des cas de fautes.

2.3.2 Classification des méthodes de génération des tests

L'auteur classe ensuite les méthodes de génération des tests selon deux critères : les critères de sélection et la technologie utilisée.

Nous utilisons l'explorateur de modèle ProB pour réaliser l'extraction des tests. Cette extraction est basée sur de la résolution de contraintes. Nous spécifions une formule (contrainte) en entrée du solveur de contrainte et ProB détermine des traces valides dans lesquelles cette formule est vérifiée.

2.3.3 Choix de la méthode d'exécution des tests

L'exécution des tests est classée selon son lien ou non avec le SUT. Pour le cas du VCI, les tests générés par le VTG sont exécutés sur l'implémentation Offline.

Notre recherche traite de l'application de la méthode VTG, qui est une méthode de test à base de modèle. Ce document nous permet de positionner notre méthode de test dans l'état de l'art des méthodes de MBT selon l'ensemble des critères proposés par l'auteur. La fin du document comporte un tableau dans lequel l'auteur classe les 119 publication selon les critères mentionnés ci-dessus. Le tableau 2.3.3 est un extrait de ce tableau dans lequel est visible la méthode VTG.

Model of System Security	Vulnerabilities
Security Model of Environment	not specified
Test Selection Criteria	Fault-Based
Maturity of Evaluated System	Production System
Evidence Measures	Effectiveness + Efficiency Measures
Evidence Level	Abstract

TABLEAU 2.1 – Classification de la méthode VTG par Felderer et al.

2.4 Positionnement

Les travaux de maîtrise [2] et de thèse [15] de Savary mettent en avant le besoin d'automatiser la génération de tests de vulnérabilité. Pour répondre à ce besoin,

l'auteur propose la méthode VTG qu'il applique à la vérification du VCI. Nous réutilisons ce même cas d'étude dans nos recherche et apportons une extension au modèle proposé permettant d'évaluer les mécanismes de vérification des instructions de branchements du VCI. L'approche combinatoire présentée dans les travaux de Calvagna et al.[5] présente un intérêt pour nos recherches car elle peut permettre de vérifier qu'une instruction de branchement respecte bien le flot d'exécution attendu.

Chapitre 3

Méthodologie

Notre recherche a pour objectif de démontrer qu'il est possible de générer automatiquement des tests de vulnérabilité à partir de la spécification d'un système sous test. Dans ce chapitre, nous établissons la méthodologie à suivre pour nous permettre d'atteindre ce but. Nous avons conçu cette méthodologie de sorte qu'elle s'applique à n'importe quel SUT.

Notre méthodologie débute par une description du cas d'étude, de laquelle découle un modèle réduit du SUT. Cette étape préliminaire au travail de modélisation permet de schématiser le modèle à réaliser avant d'en débuter la réalisation. Cela nous permet d'être confronté rapidement aux limitations que peut comporter une approche. Débute ensuite le travail de modélisation que l'on évalue suivant des critères de performance et d'adaptabilité dans un modèle complet. Cette évaluation nous permet d'effectuer une comparaison entre les expérimentations.

3.1 Description du cas d'étude

Lors de cette étape nous étudions le fonctionnement du SUT et définissons les propriétés unitaires que notre modèle final devra respecter.

Cette étude préliminaire permet de borner le travail de recherche et de débuter la modélisation sur un modèle réduit avant d'entamer une modélisation plus complète.

A l'issu de cette phase on dispose également d'un vocabulaire que l'on conserve au cours des expérimentations.

3.1.1 Description du fonctionnement global

On débute cette phase d'analyse par un diagramme d'états-transitions permettant de visualiser l'évolution du système dans le temps. Cette visualisation de haut niveau nous permet de déterminer quelles sont les conditions de passage du système d'un état à un autre. Dans le cas du VCI par exemple, nous avons déterminé que le système évolue entre 4 états : Un état d'ajout d'instruction, un état de fin de génération, un état de vérification de boucle et un état de fin de vérification de boucle.

Une fois schématisée l'évolution du système, on peut déterminer quelles variables sont nécessaires et comment elles évoluent dans le temps. Ces variables peuvent contenir des valeurs, des vecteurs de valeurs ou représenter l'évolution d'une valeur dans le temps. Cette étape permet d'éviter le doublon d'information, c'est-à-dire stocker une information qu'il n'est pas nécessaire de conserver puisqu'elle est déjà présente à un autre endroit. Dans le cas du VCI par exemple, il est nécessaire de conserver un historique des instructions exécutées de manière à pouvoir les réexecuter. Il n'est cependant pas nécessaire de stocker l'instruction en cours d'exécution car sa comparaison avec l'instruction à un temps $t - 1$ n'est pas nécessaire.

Les constantes du système peuvent également être définies lors de cette phase. Dans le cas de la modélisation du VCI, les instructions à modéliser sont des constantes. Nous choisissons de restreindre l'ensemble des instructions à modéliser à celles nous permettant de générer de tests comportant des branchements simples.

3.1.2 Définition des propriétés unitaires

Cette étape permet de formaliser des propriété contenues dans la spécification du SUT sous forme de liste d'exigences. Chaque expérimentation réalisée devra valider l'intégralité de ces propriétés pour être acceptée. Pour chaque exigence formalisée dans cette phase, on détermine une trace valide et une trace invalide. Dans le cas de la modélisation du VCI, une trace représente une suite d'instructions.

3.2 Réalisation d'une expérimentation

Dans cette section nous décrivons les étapes nécessaires à la réalisation d'une expérimentation.

Dans le cadre de notre recherche, nous avons choisi de restreindre notre champ de recherche à la génération de tests comportant des branchements simples. Pour réaliser notre objectif, nous suivons un plan en 9 étapes décrites dans les sous-sections suivantes.

A l'issue de ces 9 étapes on dispose d'un modèle fonctionnel complet à partir duquel on est en mesure d'extraire des tests de vulnérabilité concrets.

Réalisation et vérification du modèle fonctionnel

La réalisation d'un modèle fonctionnel peut se faire sur la base d'un modèle existant ou peut aborder une approche différente.

La réalisation du modèle peut utiliser une approche par raffinement. On abordera, dans cette approche, le modèle comme un empilement de plusieurs couche. La première couche est celle de plus haut niveau. Elle modélise l'évolution des variables d'état issues de l'analyse préliminaire décrite en section 3.1.1. Chaque ajout de couche se rapproche de la vision de plus bas niveau du système. La dernière couche modélisée est ainsi la plus proche de la vision concrète du système.

L'approche utilisée doit tenter d'obtenir la meilleure couverture possible. On veille cependant à ne modéliser qu'un ensemble réduit d'instructions. On diminue ainsi le temps nécessaire à la réalisation d'une expérimentation. Cet ensemble réduit doit permettre d'évaluer l'approche utilisée sur des critères d'évolutivité et de performance. On essaie par exemple de déterminer le temps nécessaire pour passer d'un modèle réduit à un modèle complet et la perte de performance que cette évolution pourrait entraîner.

Chaque couche du modèle est vérifiée par *model-checking*. Dans notre recherche nous faisons le choix d'utiliser l'explorateur de modèle ProB. Quelle que soit la méthode de vérification choisie, on documente dans cette partie le mode opératoire suivi ainsi que les paramètres utilisés.

Génération des tests fonctionnels abstraits

L'exploration du modèle fonctionnel réalisé en partie 3.2, nous permet de générer des tests fonctionnels abstraits.

ProB dispose de plusieurs modes d'extraction de tests. Dans cette partie on détermine le mode opératoire le plus approprié pour générer une suite de tests abstraits répondant aux critères de sélection déterminés en partie 3.1.2. On documente dans cette partie le mode opératoire suivi ainsi que les paramètres utilisés.

On sélectionne les tests sur des critères de couverture. Dans le cas de la modélisation du VCI, on cherche par exemple à obtenir au moins un test par instruction. Ce paramètre peut être renseigné sur ProB lors de l'extraction des tests.

A l'issue de cette phase on dispose de statistiques sur la durée nécessaire pour extraire une suite de tests.

Concrétisation des tests fonctionnels abstraits

La concrétisation des tests fonctionnels abstraits générés nécessite l'utilisation d'outils de concrétisation. Il convient donc, dans un premier temps, de trouver ou développer une librairie permettant la concrétisation des tests générés lors de nos expérimentations. Chaque modèle proposé est différent du précédent, il faut donc adapter les outils de concrétisation afin qu'ils soient applicables au modèle proposé.

A l'issue du traitement de la suite de tests abstraits par les outils de concrétisation, on obtient une suite de tests concrets exécutables sur le SUT.

Exécution des tests fonctionnels concrets sur le SUT

Les phases précédentes nous permettent de générer des tests fonctionnels concrets conformes à la spécification du SUT. Dans cette phase d'exécution, nous testons l'acceptation de ces tests par le système sous test. On s'assure ainsi que la modélisation réalisée dans les premières phases est correcte.

Dans le cas où des tests acceptables sont rejetés, on reprend l'expérimentation à la phase de réalisation du modèle puis on applique le reste de la méthodologie.

Réalisation ou adaptation du VTG

Notre recherche tend à montrer qu'il est possible d'automatiser la génération de tests de vulnérabilité. Savary et al. proposent dans [15] une chaîne d'outils VTG (*Vulnerability Test Generator (VTG)*) qui, à partir d'un modèle fonctionnel, génère une suite de tests de vulnérabilités concrets.

Si on dispose déjà d'un VTG, on réalise les modifications nécessaires au traitement du modèle fonctionnel réalisé lors de la première phase. Ces modifications peuvent aussi bien concerner l'implémentation de la chaîne d'outils VTG et la théorie VTG.

Génération des modèles mutants

Un modèle mutant, tel qu'abordé dans la partie 1.1.3, est une dérivation d'un modèle fonctionnel dans lequel on a fait varier une ou plusieurs propriétés. Le processus de génération de mutants peut être manuel ou automatique.

Génération des tests de vulnérabilité abstraits

L'objectif de cette phase est similaire à celui de la phase 3.2. Ici, les tests sont extraits à partir d'un modèle mutant et non d'un modèle fonctionnel. Les outils utilisés dans cette phase sont donc similaires à ceux utilisés lors de la phase de génération de tests fonctionnels abstraits.

Les tests extraits à l'issue de cette phase sont dits de vulnérabilité, car ils ne suivent pas la spécification du SUT.

Concrétisation des tests de vulnérabilité abstraits

Pour concrétiser les tests de vulnérabilité abstraits on réutilise les outils développés pour concrétiser les tests fonctionnels. Les tests extraits sont ici des tests de vulnérabilité que l'on peut soumettre au SUT dans le but d'analyser sa réaction.

Exécution des tests de vulnérabilité concrets sur le SUT

Cette phase conclue l'expérimentation. On réutilise la même implémentation du SUT utilisée lors de la phase 3.2. Les tests soumis au SUT sont ici tous rejetables,

puisque extraits d'un modèle non-conforme à la spécification du système sous test.

3.3 Comparaison des expérimentations

Cette partie consiste à déterminer les critères permettant de comparer chaque expérimentation et ainsi de déterminer quelle est l'approche la plus efficace pour répondre au problème donné.

Critère de performance : La performance regroupe plusieurs aspects. Dans le cadre de notre recherche, on cherche ainsi à obtenir un ensemble de tests avec le plus grand degré de couverture possible. L'approche proposée dans l'expérimentation doit également permettre de générer un nombre suffisant de tests. Une fois le modèle généré, la génération des tests doit se faire dans un temps acceptable.

Critère d'adaptabilité dans un modèle complet : Notre travail de recherche s'inscrit dans la continuité de travaux existant. Il est donc important que cette avancée puisse facilement s'intégrer aux précédentes recherches et permettent également aux travaux futurs de s'y greffer.

Critère de coût en temps humain : Le dernier critère choisi est celui du temps. Il peut concerner le temps nécessaire pour réaliser l'expérimentation ou celui de la recherche de solutions.

Chapitre 4

Vérification du VCI avec la méthode VTG

Ce chapitre présente les résultats de nos recherches. La méthodologie présentée en chapitre 3 est appliquée au cas d'étude du vérifieur de code intermédiaire de Java Card (VCI). Plusieurs approches sont présentées et comparées pour apporter une réponse au problème de la génération automatique de tests de vulnérabilité appliquée au VCI.

4.1 Description du cas d'étude

Le cas d'étude de notre recherche est le vérifieur de code intermédiaire de Java Card (VCI). Nous avons choisi d'étudier plus spécifiquement le fonctionnement du mécanisme de vérification des boucles du VCI. Ce mécanisme est déclenché par le VCI lorsqu'une boucle est détectée dans un programme.

Notre recherche se focalise sur les boucles de type "simple", c'est-à-dire des boucles dans lequel le branchement se fait dans une seule direction. Trois cas peuvent alors se présenter : le branchement en avant, le branchement en arrière et le branchement vers soit-même.

Pour s'assurer de la validité d'un programme, le VCI traite chaque instruction l'une après l'autre. Certaines instructions ont la capacité de créer des boucles simples. C'est le cas, par exemple, de l'instruction *goto*. Lors du traitement d'un *goto*, le VCI

déclenche un mécanisme de vérification qui va s'assurer de la validité du point de branchement. Il s'assure que tous les chemins d'accès au point de branchement comportent une valeur de pile qui est compatible avec l'instruction au point de branchement.

4.1.1 Description de notre approche

Notre approche consiste à modéliser chaque instruction et la réaction attendue du VCI lors de leur vérification. Les cas de tests sont obtenus en explorant le modèle avec ProB. Un programme Java Card est alors généré en ajoutant séquentiellement des instructions dans un vecteur $instructionHist$. Ce vecteur est indexé par un entier représentant le pointeur d'instruction. A l'initialisation du système, le vecteur $instructionHist$ est vide. Une fois la première instruction ajoutée par le développeur, l'index 0 du vecteur contiendra cette instruction.

La pile d'opérande est représentée par un vecteur *stack*. L'index du vecteur est un entier représentant l'ordre dans lequel les valeurs sont ajoutées. On conserve également l'historique des valeurs de *stack* dans un vecteur $stackHist$ indexé comme le vecteur $instructionHist$, c'est-à-dire par les valeurs du pointeur d'instruction.

Chaque instruction possède des pré-condition et des post-conditions en adéquation avec la spécification Java Card[12]. Lors de l'initialisation du système la pile d'opérande est vide. Il n'est donc pas possible d'exécuter d'instruction dont la post-condition serait de retirer un élément de la pile d'opérande.

Nous utilisons une variable $generationState$ pour représenter le passage du VCI dans un état de vérification de boucle. Le diagramme d'état-transition représentant ces changements d'état est représenté en figure 4.1 La présence d'une instruction de boucle dans un programme n'entraîne pas forcément la vérification de cette boucle par le VCI. Si, lors de l'ajout d'une instruction de branchement, le point de branchement contient une pile d'opérande identique à la pile courante, aucune vérification n'est nécessaire. Si la pile au point de branchement n'est pas identique, le VCI déclenche la vérification de la boucle. En cas de réussite de la vérification, le modèle termine dans l'état $verificationFinished$.

Quelle que soit l'issue de la vérification, une fois une boucle fermée il n'est plus possible d'ajouter d'instruction. Ce maintient du système dans un état où aucun

changement n'est possible est appelé «*deadlock*». Deux types de programme peuvent donc être générés : ceux qui ne contiennent pas de boucle et qui se terminent par un *return*, et ceux qui n'en contiennent qu'une.

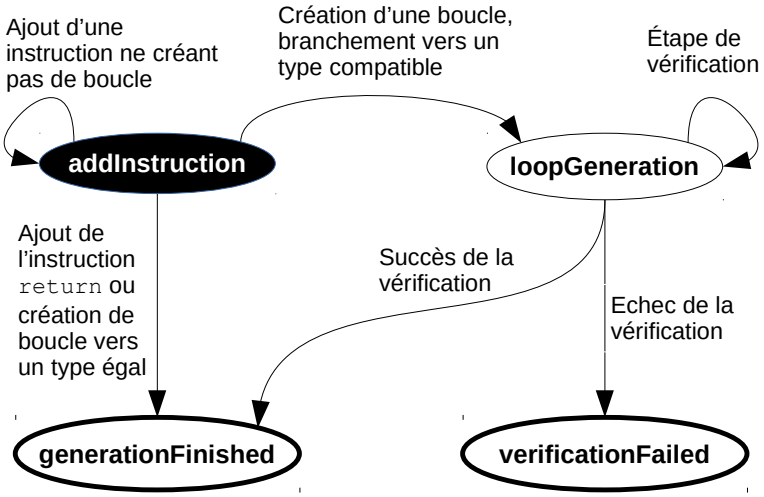

FIGURE 4.1 – Diagramme d'état-transition de notre modèle

4.1.2 Définition des propriétés unitaires

On liste dans cette partie un ensemble réduit de propriétés que notre modèle doit vérifier pour être valide. Ces propriétés sont extraites de notre analyse de la spécification Java Card.

Chaque propriété est numérotée et possède un exemple de séquence d'instructions, aussi appelée "trace". Une trace valide est une trace que le VCI est censé accepter selon la spécification Java Card.

EXI00 : Aucune instruction ne peut s'exécuter après une instruction de fin de génération (return, areturn, etc). La définition de cette propriété nous permet d'évaluer la capacité du modèle à passer d'un état de génération d'instruction à un état d'arrêt.

Traces valides :

— `return`

— `aconst_null` (ajout d'une référence nulle sur la pile)`; pop; return`

Trace invalide :

— `aconst_null; pop; return; aconst_null`

EXI01 : Lors de la création d'une boucle, la pile d'opérandes à l'index de branchement, c'est-à-dire l'index vers lequel pointe l'instruction courante, doit être de taille égale à la pile d'opérande courante. La définition de cette propriété nous permet d'évaluer la capacité du modèle à respecter des contraintes sur la taille des piles.

Trace valide :

— `goto 0`

— `aconst_null; pop; goto -2` (branchement arrière)

— `goto 3; goto -2; nop` (branchement avant)

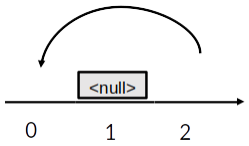

FIGURE 4.2 – Exemple de trace valide de branchement vers une pile de taille égale

Trace invalide :

— `aconst_null; pop; goto -1`

— `goto 3; goto -2; aconst_null`

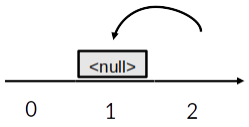

FIGURE 4.3 – Exemple de trace invalide de branchement vers une pile de taille égale

EXI02 : Lors de la création d'une boucle, la pile d'opérandes à l'index de branchement est dite "compatible" à la pile d'opérande courante, si le père commun (voir figure 1.7) entre les éléments de ces deux piles est différent de l'élément *Top*. Toutes les instructions qui succèdent à cet index doivent alors être réexecutables. La définition de cette propriété nous permet d'évaluer la capacité du modèle à respecter des contraintes sur les types contenus dans les piles.

Trace valide :

— `sconst_0; pop; bspush; goto -2`

FIGURE 4.4 – Exemple de trace valide de branchement vers une pile compatible

Trace invalide :

— `sconst_0; pop; aconst_null; goto -2`

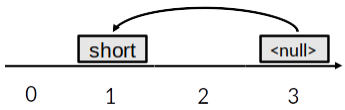

FIGURE 4.5 – Exemple de trace invalide de branchement vers une pile compatible

4.1.3 Définition des objectifs de test

L'exploration de modèles permet la génération d'un vaste ensemble de tests. Il est donc nécessaire de définir des objectifs de tests pour ne conserver que les tests les plus pertinents.

OBJ01 : Toutes les instructions doivent être utilisées au moins une fois dans un contexte linéaire.

OBJ02 : Toutes les instructions doivent être utilisées au moins une fois dans un contexte non-linéaire. La vérification des tests par le VCI doit entraîner une vérification des types.

OBJ03 : Les tests générés doivent couvrir toutes les instructions dans un contexte non-linéaire (branchement vers un type égal). La vérification des tests par le VCI ne doit pas entraîner une vérification des types.

OBJ04 : Tous les tests générés doivent être acceptés par le VCI.

4.2 Mise en application en Event-B

Dans [15], Savary et al. utilisent la méthode Event-B pour représenter la vérification par le VCI de programmes linéaires. Notre première expérimentation propose une amélioration de ce modèle en deux partie. Dans un premier temps nous réalisons un nouveau modèle de génération adapté aux programmes non-linéaires en modélisant un ensemble d'instructions contenant l'instruction *goto*. Nous sommes ainsi en capacité de générer des programmes contenant des boucles. La présence de ces boucles dans un programme déclenche dans certains cas des mécanismes de vérification de boucle par calcul d'inférence de types. Nous réalisons donc, dans un second temps, une modélisation de la vérification de boucle du VCI.

4.2.1 Réalisation et vérification du modèle fonctionnel

Le modèle développé dans cette expérimentation se compose d'une partie de génération et d'une partie de vérification. La première partie représente les instructions et l'effet de leur exécution sur le système. La seconde partie modélise les mécanismes de vérification qui ont lieu lors de la vérification de type. Cette dernière constitue la majeure partie de notre contribution à la méthode VTG. Il s'agit de générer non seulement des programmes contenant des boucles, mais de s'assurer que ces programmes déclencheront les mécanismes de vérification de boucle.

Lors de cette expérimentation nous avons établi et suivi des conventions.

ROD00 : les gardes, invariants et axiomes sont ordonnées du moins contraignant vers le plus contraignant.

ROD01 : on nomme les machines comme suit : M + numéro de raffinement + _ + nom.

Exemple : *M00_generationState* est la machine du premier niveau de raffinement (00) du modèle. Elle a pour nom *generationState*.

ROD02 : on nomme les événements comme suit : R + numéro de raffinement + *_(evt)* + nom.

Exemple : *R00_evtAddInstruction* est un événement du premier raffinement du modèle (00). Il a pour nom *AddInstruction*.

ROD03 : on nomme les gardes comme suit : *grd* + numéro de raffinement + numéro de garde. Le numéro de garde est propre à chaque événement.
Exemple : *grd0100* est la première garde (00) sur second niveau de raffinement (01).

ROD04 : on nomme les paramètres des événements comme suit :

— 'prime_' + nom : utilisé pour définir la nouvelle valeur d'une variable
— 'local_' + nom : utilisé pour faire des calculs dans l'événements
— 'arg_' + nom : argument de l'événement

ROD06 : on nomme les actions comme suit : *act* + numéro de raffinement + numéro d'action. Le numéro d'action est propre à chaque événement.
Exemple : *act0100* est la première garde (00) du second niveau de raffinement (01).

ROD07 : on nomme les contextes comme suit : *C* + numéro de contexte + _ + nom. Numéro du contexte en rapport avec le numéro du raffinement dans laquelle il est introduit.
Exemple : *C00_generationState* est le contexte du premier niveau de raffinement (00). Il a pour nom *generationState*.

ROD08 : on déclare les variables dans les contextes CXX_nom et on affecte les valeurs dans les contextes CXX_nom_prefilled (puis on importe le contexte de déclaration).
Exemple : *C14_instructions_prefilled* affecte des valeurs aux variables définies dans le contexte *C14_instructions*.

L'approche par Event-B nous permet de modéliser le VCI en plusieurs raffinements. Notre modèle final en contient 16. On peut visualiser cette approche sous forme d'une pyramide, représentée en figure 4.6, dans laquelle la base est la couche la plus abstraite, c'est-à-dire la couche commune à toutes les instructions, et le sommet la couche la plus concrète, c'est-à-dire l'instruction en elle-même.

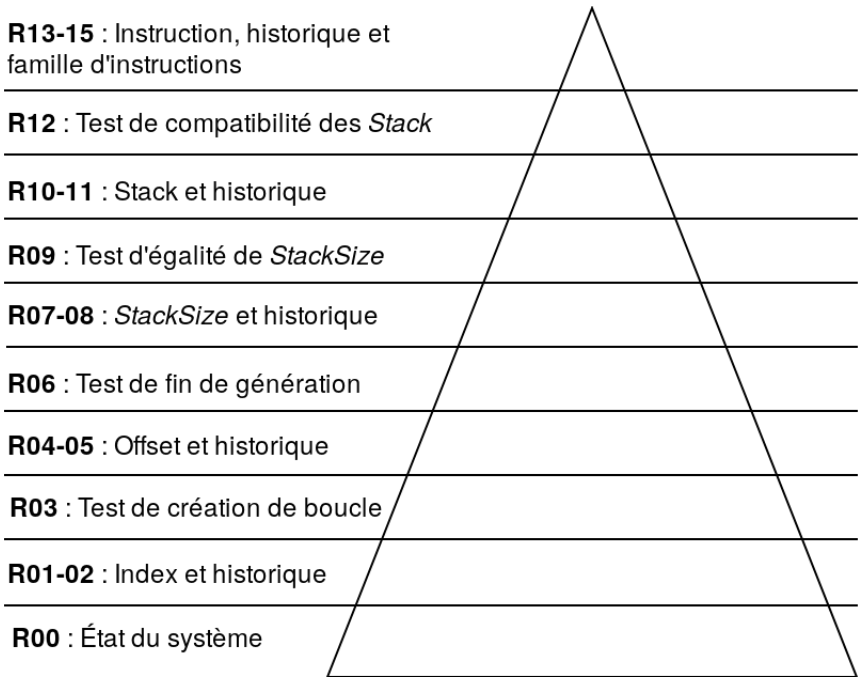

FIGURE 4.6 – Le modèle Event-B final comporte 16 niveaux de raffinements

R00 : Évolution de l'état du système

Le premier niveau de raffinement *R00* nous permet de représenter l'évolution du système. On utilise une variable *generationState* pour décrire l'état de la génération. Cette variable peut prendre 3 valeurs différentes qui constituent l'ensemble *STATES*.

addInstruction : Représente l'état initial de la génération. Le système reste dans cet état tant qu'aucune boucle n'est formée dans le programme.

loopVerification : Représente l'état de vérification d'une boucle. Si le processus de vérification est une réussite alors le système passe en *deadlock* dans l'état *generationFinished*, sinon il reste en *deadlock* dans l'état *loopVerification*. Cette expérimentation ne traite que des branchements simples. Les boucles formées sont donc des boucles infinies après lesquelles il n'est pas possible de rajouter d'autres instructions.

generationFinished : Représente l'état de fin de génération. Le système reste dans cet état en présence d'une instruction de fin de génération comme *return*.

L'événement *evtAddInstruction*, présenté ci-dessous, effectue le changement d'état vers *addInstruction*, *loopVerification* ou *generationFinished*. Il est accessible si l'état courant est à *addInstruction*. Il se situe donc dans la partie génération du modèle.

Event *R00_evtAddInstruction* $\widehat{=}$

any

prime_generationState

where

grd0000 : generationState = addInstruction

grd0001 : prime_generationState ∈ STATES

then

act0000 : generationState := prime_generationState

end

On définit également l'événement *evtLoopVerification* à ce niveau de raffinement. Il effectue le changement d'état vers *loopVerification* ou *generationFinished*. Il est accessible si l'état est à *loopVerification*. Il se situe donc dans la partie vérification du modèle.

R01-02 : Index du programme et historique

Nous introduisons la notion d'index aux raffinements *R01* et *R02*. La partie vérification de notre modèle nécessite de conserver des valeurs d'historique. On définit donc à ce niveau de raffinement deux nouvelles variables :

— **index**, tel que $index \in 0 \,..\, MaxIndex$, qui représente la position de l'instruction courante dans un programme.

— **exploredIndex**, tel que $exploredIndex \subseteq 0 \,..\, MaxIndex$, qui contient l'historique des index explorés.

La valeur de la constante *MaxIndex* est définie dans le contexte *C01*. Cette valeur est fixée à 10 pour permettre une longueur de test suffisamment grande sans altérer le temps de model-checking.

L'événement *evtAddExploredIndex*, présenté ci-dessous, est un raffinement de l'événements *evtAddInstruction*.

Event *R02_evtAddExploredIndex* $\widehat{=}$

any

prime_index

prime_exploredIndex

where

grd0100 : prime_index $\in$ 0 .. MaxIndex

grd0200 : prime_exploredIndex $\subseteq$ 0 .. MaxIndex

grd0201 : prime_exploredIndex = exploredIndex $\cup$ {index}

then

act0100 : index := prime_index

act0200 : exploredIndex := prime_exploredIndex

end

Dans la partie vérification du modèle, l'événement *evtGetIndex* est un raffinement de l'événement *evtLoopVerification*. Il borne la valeur de l'index de l'instruction en cours de vérification dans l'ensemble *exploredIndex*.

R03 : Test de création de boucle

Dans cette expérimentation, nous faisons le choix de modéliser chaque instruction de 4 manières différentes pour couvrir chaque contexte d'exécution. On sépare donc dans ce niveau de raffinement les instructions en deux groupes : celles qui forment des boucles et celles qui n'en forment pas. Ces deux groupes seront ensuite redivisés dans les niveaux de raffinements suivants.

— **NoLoop**, représente le cas où l'instruction ne crée pas de boucle et ne termine pas non plus la génération.

— **Return**, représente le cas où l'instruction ne crée pas de boucle mais termine la génération. Dans notre ensemble d'instructions à modéliser, seule l'instruction `return` correspond à ce type.

L'événement *evtLoopGeneration*, présenté ci-dessous, est un raffinement de l'événement *evtAddExploredIndex*. Aucun ajout n'est fait dans les clauses ANY et THEN, qui héritent simplement des raffinements inférieurs. La clause where est utilisée ici pour définir des contraintes permettant, ou non, l'exécution de cet événement. On définit ainsi que l'événement *evtLoopGeneration* ne peut survenir uniquement que si la nouvelle valeur d'index se trouve dans l'ensemble *exploredIndex*. L'état de génération sera alors soit à *loopVerification*, soit à *generationFinished*.

Event *R03_evtLoopGeneration* $\widehat{=}$
refines *R02_evtAddExploredIndex*

where

grd0300 : `prime_generationState` $\in$ {`loopVerification`, `generationFinish`}

grd0301 : `prime_index` $\in$ `prime_exploredIndex`

Un autre événement est défini à ce niveau de raffinement. L'événement *evtNoLoopGeneration* est un raffinement de l'événement *evtAddExploredIndex*. Il permet de couvrir deux cas : soit la génération se termine par un `return` et le système passe en état *generationFinished*, soit elle continue et le système reste en état *addInstruction*.

R04-05 : Offset et historique

L'*offset* est la différence entre l'index courant et l'index de la prochaine instruction. L'instruction *goto* prend en paramètre la valeur du saut à effectuer. Cette valeur peut être négative, nulle ou positive. La valeur d'index pointée ne peut cependant pas dépasser l'ensemble de définition $0 \mathinner{..} MaxIndex$.

Pour pouvoir rejouer les instructions lors de la vérification de boucle, il est nécessaire de connaître les valeurs de sauts. On conserve les différentes valeurs d'offset dans un vecteur *offsetHist*.

Deux nouvelles variables sont donc créées à ce niveau de raffinement :

— **offset**, tel que $offset \in -MaxOffset \mathinner{..} MaxOffset$

— **offsetHist**, tel que $offsetHist \in (exploredIndex \rightarrow -MaxOffset..MaxOffset)$

La valeur de la constante *MaxOffset* est définie dans le contexte *C04*. Cette valeur est fixée à 10. Il n'est en effet pas nécessaire de disposer d'une valeur plus élevée pour

générer des sauts étant donné que la valeur d'index est bornée à 10 par la constante *MaxIndex*.

L'événement *evtLoopAddOffsetHist*, présenté ci-dessous, est un raffinement de l'événement *evtLoopGeneration*.

Event *R05_evtLoopAddOffsetHist* $\widehat{=}$

any

prime_offset

prime_offsetHist

where

grd0400 : $prime\_offset \in -MaxOffset \mathinner{..} MaxOffset$

grd0401 : $prime\_offset = prime\_index - index$

grd0500 : $prime\_offsetHist = offsetHist \cup \{index \mapsto prime\_offset\}$

then

act0400 : $offset := prime\_offset$

act0500 : $offsetHist := prime\_offsetHist$

end

L'événement *evtNoLoopAddOffsetHist* est un autre événement défini à ce niveau de raffinement. Il possède les mêmes gardes que l'événement *evtLoopAddOffsetHist* mais couvre le cas où l'instruction ne crée pas de boucle.

Dans la partie vérification du modèle, l'événement *evtGetOffset* raffine l'événement *evtGetIndex*. Il récupère la valeur d'offset depuis la variable *offsetHist*.

R06 : Test de fin de génération

Dans ce niveau de raffinements, on sépare les instructions en deux groupes : celles qui terminent la génération et celles qui ne la terminent pas.

L'événement *evtLoopFinishGeneration*, présenté ci-dessous, est un raffinement de l'événement *evtLoopAddOffsetHist*. Comme pour l'événement *R03_evtLoppGeneration*,

ici les clauses ANY et THEN ne sont pas modifiés et héritent simplement des raffinements inférieurs. La clause WHERE définit que l'état du système sera à *generationFinished* ou *loopVerification* selon le cas de création de boucle spécifié dans les raffinements suivants.

Event *R06_evtLoopFinishGeneration* $\widehat{=}$
refines *R05_evtLoopAddOffsetHist*

where

grd0600 : prime_generationState $\in$ {generationFinished, loopVerification}

On définit également l'événement *evtNoLoopNoFinishGeneration* à ce niveau de raffinement. Il change l'état du système à l'état *addInstruction* pour que l'ajout des instructions dans le programme puisse se poursuivre.

R07-08 : *StackSize* et historique

La fermeture d'une boucle ne peut survenir que si la pile d'opérandes courante est égale ou compatible à la pile à l'index de branchement. Dans les deux cas, cela suppose que ces deux piles seront de taille égale. Il est donc nécessaire de conserver une information d'historique sur la taille des piles.

Deux nouvelles variables sont introduites à ce niveau de raffinement :

— **stackSize**, tel que $stackSize \in 0 \,..\, MaxStackSize$.

— **stackSizeHist**, tel que $stackSizeHist \in (exploredIndex \rightarrow 0..MaxStackSize)$

L'événement *evtLoopChangeStackSizeHist*, présenté ci-dessous, est un raffinement de l'événement *evtLoopFinishGeneration*. Il ajoutent le singleton $\{index \mapsto stackSize\}$ dans la variable *stackSizeHist*.

Event *R08_evtLoopChangeStackSizeHist* $\widehat{=}$

any

prime_stackSize

prime_stackSizeHist

where

grd0700 : prime_stackSize $\in$ 0 .. MaxStackSize

grd0800 : prime_stackSizeHist = stackSizeHistU{index ↦ stackSize
}

then

act0700 : stackSize := prime_stackSize

act0800 : stackSizeHist := prime_stackSizeHist

end

Dans la partie génération, l'événement *evtNoLoopChangeStackSizeHist* possède les mêmes gardes que *evtLoopChangeStackSizeHist* mais couvre le cas où on modifie la pile d'opérandes sans créer de boucle.

Dans la partie vérification, l'événement *evtGetStackSize* récupère la valeur de *stackSize* depuis *stackSizeHist*.

R09 : Test d'égalité de *StackSize*

À ce stade, les instructions sont divisées en trois groupes : celles qui finissent la génération, celles qui changent de taille de la pile d'opérandes et celles qui créent une boucle vers une pile d'opérandes de taille égale.

L'événement *evtLoopEgalStackSize*, présenté ci-dessous, est un raffinement de l'événement *evtLoopChangeStackSizeHist*. Il évalue si la taille de la pile à l'index courant est identique à la taille de pile à l'index précédent.

Event *R09_evtLoopEgalStackSize* ≙
refines *R08_evtLoopChangeStackSizeHist*

any

...

where

grd0900 : prime_stackSize = prime_stackSizeHist(prime_index)

then

... : ...

end

Le cas où taille de la pile à l'index courant n'est pas identique à la taille de la pile à l'index précédent est couvert par l'événement *evtNoLoopChangeStackSizeHist* qui est ici raffiné sans ajout de gardes.

R10-11 : *Stack* et historique

Pour distinguer le cas où le branchement s'effectue vers une pile d'opérandes égale du cas où le branchement s'effectue vers une pile compatible, il est nécessaire de définir le contenu de la pile d'opérandes et son historique. Entre les niveaux de raffinements 10 et 11, on définit donc deux nouvelles variables :

- **stack**, tel que $stack \in 0..MaxStackSize \rightarrow TYPES$, avec $TYPES$ l'ensemble des types pouvant être contenu dans une *stack*.
- **stackHist**, de type $(0..MaxIndex \rightarrow (0..MaxStackSize \rightarrow TYPES))$.

L'événement *evtLoopChangeStack*, présenté si dessous, est un raffinement de l'événement *evtLoopEgalStackSize*. Il permet la création d'une boucle dans laquelle la pile d'opérandes est modifiée, soit pour contenir des types, soit pour devenir vide. Ces deux cas sont couverts respectivement par les gardes *grd1000* et *grd1001*. On crée ici une distinction entre le cas de la création de boucle vers des types égaux de celle vers des types inégaux. Dans ce second cas de création de boucle, on remplace le contenu de la pile d'opérande par une nouvelle pile, notée $arg\_resultInference$ et de type $0..MaxStackSize \rightarrow TYPES$. Le contenu de cette nouvelle pile n'est pas précisé à ce niveau de raffinement. La seule définition faite ici est que son contenu est écrit dans l'historique des piles $stackHist$ à l'emplacement de la pile qui se situait à l'index de branchement $prime\_index$. Si on réalise un branchement à l'index courant, tel que couvert par la garde *grd1101*, alors aucune réécriture n'est nécessaire, on ne fait qu'ajouter la pile $arg\_resultInference$ dans l'historique.

Event *R11_evtLoopChangeStack* $\widehat{=}$

any

```
prime_stack
arg_resultInference
prime_stackHist
```

where

grd1000 : prime_stackSize $\neq 0 \Rightarrow$
(prime_stack $\in 0 \mathinner{..} \text{prime\_stackSize} - 1 \rightarrow$ TYPES)

grd1001 : prime_stackSize $= 0 \Rightarrow$ prime_stack $= \varnothing$

grd1100 : arg_resultInference $\in 0 \mathinner{..} \text{prime\_stackSize} - 1 \rightarrow$ TYPES

grd1101 : prime_index $\neq$ index $\Rightarrow$
prime_stackHist $=$ (stackHist $\lhd\!\!+$
{prime_index $\mapsto$ arg_resultInference}) $\cup$ {index $\mapsto$ stack}

grd1102 : prime_index $=$ index $\Rightarrow$
prime_stackHist $=$ stackHist $\cup$
{prime_index $\mapsto$ arg_resultInference}

then

act1000 : stack $:=$ prime_stack

act1100 : stackHist $:=$ prime_stackHist

end

La partie génération du modèle définit également l'événement *evtNoLoopChangeStack* qui possède les gardes *grd1000* et *grd1001* communes à *evtLoopChangeStack* pour définir le contenu de la pile d'opérandes selon qu'elle soit vide ou non. Cet événement ne crée pas de boucle et ajoute donc simplement le singleton $\{index \mapsto stack\}$ dans *stackHist*.

Dans la partie vérification du modèle, l'événement *evtGetStack* vérifie si la pile d'opérandes à l'index courant dans *stackHist* est compatible avec la pile de l'instruction suivante.

R12 : Test de compatibilité des *Stack*

A ce stade, on sépare les instructions qui créent des branchements en deux : celles qui branchent vers une pile égale et celles qui branchent vers une pile compatible. En effet, dans le premier cas la vérification de la boucle est inutile puisque la pile n'a pas été modifiée. Cette vérification est nécessaire dans le second cas.

L'événement *evtLoopChangeStack_Comp*, présenté ci-dessous, est un raffinement de l'événement *evtLoopChangeStack*. Il intervient si le branchement s'effectue vers une

pile d'opérandes dont les types sont compatibles entre eux. On déclare une constante *Inference* qui contient le treillis des types, tel que $Inference \in POW(TYPES) \leftrightarrow TYPES$. Le symbole $\leftrightarrow$ est défini comme suit. Soit A et B deux ensembles : $A \leftrightarrow B = POW(A \times B)$, où $POW(A)$ dénote l'ensemble de tous les sous-ensembles de A. Pour vérifier leur compatibilité, on vérifie que toutes les paires de types appartiennent à l'ensemble *Inference*. Comme pour l'événement *evtLoopChangeStack*, deux cas se distinguent selon l'index de branchement, couverts par les gardes *grd1202* et *grd1203*. Si on réalise le branchement vers l'index courant, alors la paire de types cherchée dans l'ensemble *Inference* se compose du type dans la pile à l'index de branchement et de celui dans la pile située à l'index courant. Sinon, elle se compose du type dans la pile à l'index de branchement et de celui dans la pile située dans l'historique.

Event *R12_evtLoopChangeStack_Comp* $\widehat{=}$
refines *R11_evtLoopChangeStack*

any

...

where

grd1200 : $\texttt{prime\_generationState} = \texttt{loopGeneration}$

grd1201 : $\texttt{prime\_stackSize} > 0$

grd1202 : $\texttt{prime\_index} \neq \texttt{index} \Rightarrow$
$\forall \texttt{i} \cdot \texttt{i} \in \texttt{dom}(\texttt{stackHist}(\texttt{prime\_index})) \Rightarrow$
$\texttt{arg\_resultInference}(\texttt{i}) = \texttt{Inference}($
$\{\texttt{prime\_stack}(\texttt{i}), \texttt{stackHist}(\texttt{prime\_index})(\texttt{i})\})$

grd1203 : $\texttt{prime\_index} = \texttt{index} \Rightarrow$
$\forall \texttt{i} \cdot \texttt{i} \in \texttt{dom}(\texttt{stackHist}(\texttt{prime\_index})) \Rightarrow$
$\texttt{arg\_resultInference}(\texttt{i}) = \texttt{Inference}($
$\{\texttt{prime\_stack}(\texttt{i}), \texttt{stack}(\texttt{i})\})$

then

... : ...

end

L'événement *evtLoopChangeStack_Egal* intervient si le branchement s'effectue vers une pile égale à la pile courante. Tous les types de la pile précédente doivent être identiques.

R13-15 : Instruction, historique et famille d'instruction

L'ensemble des instructions nécessaires à l'étude des branchements simples peut se diviser en trois sous-ensembles. On nomme *normalInstructions* l'ensemble des instructions qui ne créent pas de boucles et qui ne terminent pas la génération. L'ensemble *loopInstructions* contient les instructions qui créent des boucles. Les instructions qui terminent la génération sont placées dans un ensemble *returnInstructions*

normalInstructions = {aconst_null, bspush, nop, pop, pop2, ret, s2b, sadd, sand, sconst_m1, sconst_0, sconst_1, sconst_2, sconst_3, sconst_4, sconst_5, sdiv, smul, sneg, sor, srem, sshl, sshr, sspush, ssub, sushr, sxor}

loopInstructions = {goto, goto_w}

returnInstructions = {return}

On limite notre ensemble final aux instructions nous permettant de vérifier les propriétés unitaire définies en sous-section 4.1.2. Cette limitation nous permet d'obtenir rapidement un modèle fonctionnel et d'appréhender les limites éventuelles de notre approche.

Au niveau de raffinement *R13*, on sépare les instructions suivant leur action sur la pile d'opérandes : Ajout d'un ou deux éléments, retrait d'un ou deux éléments, etc.

- Les événements suffixés par *_noChange* ne modifient pas la pile entre l'index précédent et l'index courant.
- Les événements suffixés par *_pop1* retirent un élément de la pile.
- Les événements suffixés par *_pop1_push1* retirent et ajoutent d'un élément de la pile.
- Les événements suffixés par *_pop2* retirent deux éléments de la pile.
- Les événements suffixés par *_pop2_push1* retirent deux éléments et ajoutent un élément sur la pile.
- Les événements suffixés par *_push1* ajoutent un élément sur la pile.

Aux niveau de raffinement *R14*, on introduit la variable *instruction*, tel que $instruction \in INSTRUCTIONS$. Pour rappel, on modélise chaque instruction de 4 manières différentes suivant son action dans le programme.

- **LoopEqual (LE)** représente une instruction pour laquelle l'index de la prochaine instruction est un index déjà exploré dans lequel la pile d'opérandes est égale à la pile courante. Ce type d'instruction ne nécessite pas de calcul d'inférence.
- **LoopCompatible (LC)** représente une instruction pour laquelle l'index de la prochaine instruction est un index déjà exploré et dans laquelle la pile est compatible à la pile courante. Ce type d'instruction nécessite un calcul d'inférence.
- **NoLoop (NL)** représente une instruction qui ne crée pas de boucle.
- **GetStack (GS)** représente une instruction rééxecutée lors de la vérification de boucle.

Au niveau de raffinement *R15*, on introduit la variable *instructionHist* tel que $instructionHist \in (0 \mathinner{..} MaxIndex \pfun INSTRUCTIONS)$. Chaque événement précédent est raffiné pour ajouter l'instruction courante dans un vecteur *instructionHist*.

L'événement *LC_smul*, présenté ci-dessous, contient la spécification de l'instruction *smul* dans le contexte *LoopComp*. D'après la spécification Java Card, l'instruction *smul* retire deux éléments de type *short* au sommet de la pile courante, les multiplie entre eux et pousse un résultat de type *short* au sommet de la pile. On renomme le type *short* en *shart* pour qu'il ne soit pas interprété comme *sh or t* par Event-B.

Event *R15_LC_smul* $\widehat{=}$
refines *R14_LC_smul*
refines *R13_evtLoopComp_pop2_push1*

any

`prime_instruction`

`prime_instructionHist`

where

grd1300 : $\texttt{stackSize} \geq 2$

grd1301 : $\texttt{prime\_stackSize} = \texttt{stackSize} - 1$

grd1400 : prime_instruction = smul

grd1401 : stack(stackSize − 1) = shart

grd1402 : stack(stackSize − 2) = shart

grd1403 : prime_stack(prime_stackSize − 1) = shart

grd1404 : prime_offset = 1

grd1500 : prime_instructionHist = instructionHist ∪
{index → prime_instruction}

then

act1400 : instruction := prime_instruction

act1500 : instructionHist := prime_instructionHist

end

Vérification du modèle

Comme énoncé en chapitre 3, il existe différentes façons de vérifier un modèle. La première est la vérification par rapport à des propriétés unitaires. Nous vérifions donc que notre modèle respecte les propriétés énoncées en partie 4.1.2. Pour chaque propriété, on vérifie que les traces valides peuvent bien s'exécuter et que le modèle ne permet pas d'exécuter les traces invalides. On collecte les résultats de nos vérifications dans le tableau 4.1.

Propriété	Résultat de la vérification
EXI00 : Pas d'instruction après `return`	OK
EXI01 : Taille de piles égales lors du branchement	OK
EXI02 : Vérification de boucle si types compatibles	OK

TABLEAU 4.1 – Vérification du modèle selon les propriétés unitaires de la section 4.1.2

Une seconde méthode de vérification consiste à explorer le modèle et effectuer un *model-checking*. Cette méthode nous permet de vérifier de manière automatique que notre modèle respecte bien tous ses invariants. Nous effectuons ce *model-checking* à l'aide du plugin ProB de Rodin sur chaque machine de notre modèle. On collecte des statistiques dans le tableau 4.2. On remarque qu'une fois dépassé le raffinement

7 nous manquons de mémoire pour réaliser un *model-checking* complet. L'ajout des variables d'historique, d'abord pour l'offset puis pour stackSize, semble provoquer une explosion combinatoire.

Machine	Temps (en secondes)
M00_generationState	<1
M01_index	<1
M02_exploredIndex	1
M03_loopNoLoop	2
M04_offset	6
M05_offsetHist	465
M06_finishNoFinish	510
M07_stackSize	1288
M08_stackSizeHist	>1800
M09_egalStackSize	>1800
M10_stack	>1800
M11_stackHist	>1800
M12_branchCompStack_loopVerification	>1800
M13_stackPushPop	>1800
M14_instructions	>1800
M15_instructionsHist	>1800

TABLEAU 4.2 – Résultats du *model-checking* du modèle.

La vérification par rapport à une implémentation de référence est une dernière méthode de vérification du modèle. Nous utilisons l'implémentation d'Oracle du Java Card Bytecode Verifier comme implémentation de référence pour ce SUT. Cette technique de vérification nécessite d'avoir des tests fonctionnels au format CAP. Le processus suivi pour générer et concrétiser nos tests fonctionnels abstraits est décrit dans la sous-section suivante.

4.2.2 Génération des tests fonctionnels abstraits

L'exploration du modèle fonctionnel réalisé en sous-section précédente nous permet d'extraire des tests fonctionnels abstraits. Les tests abstraits correspondent à des traces d'exécution du modèle. ProB dispose de plusieurs outils pour extraire ces traces. Nous faisons le choix d'utiliser le générateur de test à base de contraintes,

ou *Constraint-based test case generation (CBC)*. Cet outil de génération de test nous permet d'extraire des traces qui vérifient un prédicat donné. Il nous permet également de définir une longueur maximale d'exploration. On débute la génération des tests à la longueur 1 puis on augmente cette valeur jusqu'à couvrir l'ensemble des événements. L'outil CBC nous génère en sortie un fichier XML dont l'extrait est présenté en programme 4.1.

```
<test_case>
    <initialisation></initialisation>
    <step></step>
    <step></step>
    ...
    <step></step>
</test_case>
```

Programme 4.1 – Structure du fichier XML de tests abstraits

.

Profondeur max.	Nb. de tests	Couverture (%)	Temps d'extract. (secondes)
1	3	3	5
2	3	3	74
3	19	17	189
4	26	23	2775
5	74	65	**49776**

TABLEAU 4.3 – Extraction des tests abstraits du premier modèle Event-B

Comme indiqué au tableau 4.3, ce modèle ne nous permet pas d'extraire des tests d'une longueur supérieure à 5. Une longueur aussi faible ne permet pas de couvrir les traces de vérification. On se retrouve ici face à un phénomène d'explosion combinatoire. Lors de l'extraction des tests de longueur 6, l'espace d'état est trop important pour pouvoir être stocké sur la mémoire de l'ordinateur mis à disposition pour nos tests. On estime qu'obtenir une suite de tests de longueur 6 à partir de ce modèle nécessiterait 6 jours et demi. Nous choisissons donc de créer un second modèle avec un nombre réduit d'instructions afin de limiter l'espace d'états. Les instructions qui ne sont pas nécessaires à la vérification des propriétés unitaires énoncées en section

4.1.2 sont ainsi retirées. Les instructions modélisées dans ce second modèle allégé se limitent donc à l'ensemble *{aconst_null, bspush, goto, pop, sconst_0, smul, return}*. Les deux modèles sont identiques jusqu'aux raffinements M14 et M15.

Profondeur max.	Nb. de tests	Couverture (%)	Temps d'extract. (secondes)
1	2	8	1
2	2	8	6
3	7	32	8
4	9	40	23
5	18	76	62
6	19	80	283
7	24	100	1812

TABLEAU 4.4 – Extraction des tests abstraits du second modèle Event-B

Le tableau 4.4 présente les résultats de l'extraction des tests abstraits à partir du second modèle Event-B. On constate que la réduction du nombre d'instruction a permis d'extraire des tests de longueur 7. Le nombre de tests générés est moins important mais ils permettent de couvrir 100% des instructions modélisées en 1812 secondes. De plus, l'objectif du test à base de contrainte est de générer un nombre minimal de tests couvrant les contraintes qui lui sont données en entrée. Compte tenu de ces résultats, nous décidons de poursuivre l'expérimentation sur ce second modèle uniquement.

4.2.3 Concrétisation des tests fonctionnels abstraits

Pour concrétiser les tests, nous avons développé un outils de concrétisation de tests abstraits. Le fonctionnement et l'implémentation de cet outil sont détaillés en chapitre 5.

Cet outil réalise une analyse du fichier XML obtenu à l'étape précédente pour en extraire des fichiers JCA. Chaque *<step>* du fichier XML correspond à une instruction que l'outil fait passer de sa représentation abstraite à sa représentation concrète. Nous nous aidons de la spécification Java Card [12] pour obtenir l'implémentation de chaque

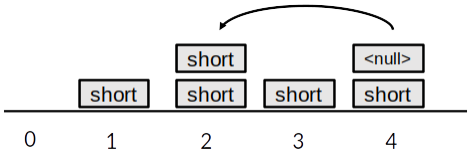

FIGURE 4.7 – Visualisation du test rejeté

instruction. Pour obtenir l'implémentation JCA des instructions de branchement, nous réalisons une étude de l'équivalence entre des instructions Java et leur équivalent JCA dont le résultat est présenté en annexe A. Un extrait est présenté en sous-section 1.3.2.

À l'issue de cette étape nous obtenons 24 tests fonctionnels concrets sous forme de fichier JCA.

4.2.4 Exécution des tests fonctionnels sur le VCI

Nous convertissons les fichiers JCA obtenus à l'étape précédente en fichiers CAP afin qu'ils puissent être exécutés sur le VCI. Nous avons pour cela développé un outil de conversion automatique qui utilise la librairie *capgen* de Java Card.

Les fichiers CAP obtenus sont ensuite exécutés sur le VCI à l'aide d'un second outil que nous avons développé. Il utilise ensuite la librairie *offcardverifier* de Java Card et donne le résultat de l'exécution de chaque test sur le VCI. Si le test échoue, le vérifieur affiche la raison de son rejet dans la console.

Le tableau 4.5 présente les résultats de la vérification par le VCI des tests concrets fonctionnels. Sur 24 tests générés, 23 ont été acceptés et 1 a été refusé par le VCI avec l'erreur suivante : *Erreur de type à un point de jointure sur l'élément de pile numéro 1 : short, <null>*. La figure 4.7 montre l'incompatiblité de la pile à l'index 4 avec la pile à l'index 2, vers laquelle l'instruction *goto* essaie de brancher.

Le fait qu'il soit possible de générer ce test invalide à partir de notre modèle fonctionnel nous démontre qu'il existe des erreurs dans les gardes qui définissent le branchement vers un type compatible. Il faudrait donc revoir les gardes de chaque événements de type LC à partir du raffinement *R12*. Comme expliqué en partie 4.2.2, cette expérimentation met en lumière les limitations du langage Event-B pour répondre

Identifiant	Trace	VCI
CAP01	goto 0	OK
CAP02	return	OK
CAP03	aconst_null, goto 0	OK
CAP04	bspush, goto 0	OK
CAP05	goto 1, goto 0	OK
CAP06	sconst_0, goto 0	OK
CAP07	aconst_null, goto 0	OK
CAP08	aconst_null, pop, goto -2	OK
CAP09	goto 3, goto -2, aconst_null, pop	OK
CAP10	sconst_0, sconst_0, smul, goto -2	OK
CAP11	aconst_null, aconst_null, pop, goto -2	OK
CAP12	aconst_null, aconst_null, pop, goto -2	OK
CAP13	aconst_null, goto 2, pop, goto -2, aconst_null	OK
CAP14	aconst_null, pop, aconst_null, goto -2	OK
CAP15	aconst_null, sconst_0, pop, goto -2	OK
CAP16	bspush, goto 3, goto -2, pop, bspush	OK
CAP17	sconst_0, goto 3, goto -2, pop, sconst_0	OK
CAP18	sconst_0, goto 3, goto -2, sconst_0, smul	OK
CAP19	aconst_null, goto 2, pop, goto -2, aconst_null	OK
CAP20	aconst_null, goto 2, aconst_null, goto -2, pop	OK
CAP21	bspush, goto 3, goto -2, pop, bspush	OK
CAP22	sconst_0, sconst_0, smul, aconst_null, goto -2	NOK
CAP23	sconst_0, goto 3, goto -2, pop, sconst_0	OK
CAP24	sconst_0, goto 3, goto -2, sconst_0, smul	OK

TABLEAU 4.5 – Résultats de la vérification par le VCI des tests concrets fonctionnels

au problème de la vérification de type. Chaque opération de vérification entraîne en effet un coût d'une action lors de la génération des tests par ProB. Nous choisissons de poursuivre l'expérimentation malgré les erreurs constatées sur notre modèle et les limitations du langage Event-B pour poursuivre notre analyse de l'applicabilité de la méthode VTG à l'Event-B.

4.2.5 Génération manuelle des modèles mutants

La chaîne d'outils du VTG a été développée par Savary et al. en 2015 sous l'API ProB 1.0. Elle se compose de deux parties : l'une réalisant un aplanissement du modèle fonctionnel et l'autre générant des modèles mutants à partir du modèle aplani initial. Cette chaîne d'outils n'est pas compatible avec la version actuelle de l'API ProB. Son adaptation à la nouvelle version de l'API ProB nécessiterait plusieurs mois. Elle n'a donc pas pu être utilisée pour notre travail de recherche.

Nous réalisons la génération des modèles mutants à la main en suivant l'algorithme 1 de la sous-section 1.1.4. Les événements à muter sont ceux qui représentent la partie de vérification, c'est-à-dire ceux préfixés par *GS*. On remplace chaque garde de ces événements par deux gardes *grd* et *grd_t* représentant respectivement la conjonction des gardes que l'on souhaite conserver et la conjonction des gardes que l'on souhaite transformer.

L'événement *GS_smul* possède les gardes suivantes :

Event *R15_GS_smul* $\widehat{=}$

$\texttt{grd1401} : \texttt{stack}(\texttt{stackSize} - 2) = \texttt{shart}$

$\texttt{grd1402} : \texttt{stack}(\texttt{stackSize} - 1) = \texttt{shart}$

$\texttt{grd1403} : \texttt{prime\_stack}(\texttt{prime\_stackSize} - 1) = \texttt{shart}$

On souhaite évaluer les mécanismes de vérification de types du VCI. On va donc transformer les gardes qui concernent les types et conserver toutes les autres. L'ensemble des mutations générables est égal à $neg(grd\_t)$, avec grd_t tel que

$\texttt{grd\_t} : \texttt{stack}(\texttt{stackSize} - 2) = \texttt{shart} \wedge$

$\texttt{stack}(\texttt{stackSize} - 1) = \texttt{shart} \wedge$

$\texttt{prime\_stack}(\texttt{prime\_stackSize} - 1) = \texttt{shart}$

On applique ensuite les règles de mutation décrites dans 1.1.3 à *grd_t*. Chaque mutation obtenue constitue un nouvel événement qui est ensuite ajouté dans un modèle distinct. Pour repérer les événements ayant subi une mutation, nous ajoutons le suffixe *_mut* suivi d'un numéro. Nous ajoutons ce même suffixe aux modèles transformés.

Event *R15_GS_smul_mut00* $\widehat{=}$

grd1401 : stack(stackSize − 2) = shart

grd1402 : stack(stackSize − 1) = **referenceTypes**

grd1403 : prime_stack(prime_stackSize − 1) = shart

Event *R15_GS_smul_mut01* $\widehat{=}$

grd1401 : stack(stackSize − 2) = **referenceTypes**

grd1402 : stack(stackSize − 1) = shart

grd1403 : prime_stack(prime_stackSize − 1) = shart

Event *R15_GS_smul_mut02* $\widehat{=}$

grd1401 : stack(stackSize − 2) = **referenceTypes**

grd1402 : stack(stackSize − 1) = **referenceTypes**

grd1403 : prime_stack(prime_stackSize − 1) = shart

Une fois ces 3 événements mutants intégrés dans des modèles distincts, nous obtenons 3 modèles mutants.

4.2.6 Génération et concrétisation des tests de vulnérabilité abstraits

La génération des tests de vulnérabilité abstraits suit le même protocole que celui décrit en partie 4.2.2. Les objectifs de test sont également communs. Nous souhaitons générer des tests faisant intervenir des mécanismes de vérification de boucle. Cependant, dans le cas de la génération de test de vulnérabilité abstraits, on souhaite également que nos tests couvrent l'instruction ayant subi une mutation, c'est-à-dire ceux suffixés par *_mut*.

Modèle	Nombre de tests	Temps d'extraction (sec.)
MUT00	1	363.25
MUT01	0	3712.31
MUT02	0	3740.11

TABLEAU 4.6 – Génération des tests de vulnérabilité abstraits

Comme indiqué en tableau 4.6, seul le modèle mutant *MUT00* nous permet d'extraire un test de vulnérabilité abstrait au format XML. La structure de ce test est similaire à celle obtenue après l'étape 4.2.2. Il nous est donc possible de réutiliser l'outil développé lors de la phase 4.2.3. A l'issue de cette étape on dispose donc d'un test de vulnérabilité.

4.2.7 Exécution des tests de vulnérabilité concrets sur le VCI

L'exécution du test de vulnérabilité utilise le même outil que celui développé en sous-section 4.2.4.

Identifiant	Trace	VCI
MUT00_CAP00	sconst_0, sconst_0, smul, aconst_null, goto -2	NOK

TABLEAU 4.7 – Vérification par le VCI du test de vulnérabilité concret

Le tableau 4.7 présente le résultat de la vérification par le VCI du test de vulnérabilité. Le test de vulnérabilité généré est le même que celui généré lors de la phase de génération des tests fonctionnels. Comme expliqué en conclusion de la section 4.2.4, nous avons constaté que les gardes du modèle fonctionnel ne sont pas assez strictes. Cette erreur a provoqué la génération de tests qui représentent en réalité un comportement fautif. Il est donc normal de retrouver un test similaire lors de la phase de génération des tests de vulnérabilité. Le test soumis au VCI lors de cette phase a été rejeté par le VCI pour la raison suivante : *Erreur de type à un point de jointure sur l'élément de pile.*

4.2.8 Statistiques et observations

Cette première expérimentation en Event-B s'est basée sur les travaux de Savary et al. Une partie de la chaîne d'outil du VTG n'a cependant pas pu être utilisée car incompatible avec la version actuelle de l'API ProB.

Notre modèle final est l'aboutissements de 4 mois de travail entre janvier et avril 2016 dans lequel plusieurs mini-modèles ont été réalisés. Le mois de janvier a été dédié à la mise en place des conventions sur Rodin et à la réalisation d'une dizaine d'exercices de familiarisation avec Rodin. La réalisation d'un modèle de génération d'instruction sans création de boucle a débuté en février et a duré 2 semaines. Ce modèle comporte 10 niveaux de raffinements. L'ajout de la création de boucle et de la vérification a débuté mi-février et a duré 3 semaines. Ce second modèle constitue notre modèle complet initial. Il modélise 30 instructions. Lors de l'exécution, le calcul des gardes et des événements accessibles s'effectue dans un délais inférieur à 1 seconde du début à la fin. Pour le réaliser nous avons développé 4 outils présentés en chapitre 5.

La génération de tests à partir de notre modèle a débuté mi-mars 2016 et a duré 6 semaines. Nous avons développé les outils de concrétisation et d'exécution lors de cette partie de l'expérimentation. Du fait de la taille du modèle, il ne nous a pas été possible de générer des tests de longueur supérieure à 5. Nous avons donc décidé de poursuivre l'expérimentation sur une version allégée du modèle, limitée à 7 instructions. Cette version allégée nous a permis de générer des tests de longueur 7 dans un délai raisonnable. Nous aurions également pu réduire le nombre d'instruction à couvrir lors du CBC. Cependant, comme le montre le tableau 4.8, les performances sont très inférieures à celles obtenues en retirant les instructions du modèle.

Profondeur	Couverture (%)	Modèle réduit (sec.)	Modèle complet (sec.)
1	9	1	1
2	9	6	61
3	33	8	88
4	42	23	637
5	76	**42**	**4856**

TABLEAU 4.8 – Couverture des instructions du modèle réduit sur le modèle complet.

Les mutations ont été réalisés à la main. Nous en avons réalisé 3 en 10 minutes. La structure des modèles mutants ne diffère pas de celle du modèle fonctionnel. Aucune adaptation des outils d'extraction, de concrétisation et d'exécution n'a donc été requise. L'exécution des tests fonctionnels et des tests de vulnérabilité sur le VCI se fait instantanément.

4.3 Mise en application en B classique

La précédente expérimentation nous a permis d'appréhender les avantages et inconvénients que comportent l'utilisation de la méthode Event-B pour la génération de tests de vulnérabilité.

Pour modéliser la vérification de type du VCI, nous avons utilisé des événements spécifiques. Pour chaque instruction du modèle, il fallait ainsi définir une instruction de vérification (*GetStack*), ce qui doublait la taille des traces du modèle. Cette méthode n'est pas viable pour générer des traces permettant de faire intervenir le mécanisme de vérification de boucle du VCI.

Le langage B dispose d'une fonction *while*. Il est ainsi possible de réaliser toute la réexécution des instructions en utilisant un seul événement, appelé "action" dans le langage B. La longueur finale d'une trace de test de longueur n est donc de $n + 1$.

4.3.1 Réalisation et vérification du modèle fonctionnel

L'approche utilisée dans cette expérimentation reprend l'ensemble des instructions utilisées dans la version allégée de 4.2, à savoir l'ensemble *{aconst_null, bspush, goto, pop, sconst_0, smul, return}*. Il sera ainsi possible de comparer les résultats avec ceux obtenus à l'issu de la première expérimentation.

Pour cette expérimentation, nous faisons le choix de ne pas utiliser de raffinements. Ce choix simplifie notre modèle et nous permet de réaliser rapidement une preuve de concept. En contrepartie, cette approche par couche unique réduit considérablement les possibilités d'intégration de notre modèle dans un modèle global.

On réalise également une réduction du nombre de variables. Les variables contenant des informations similaires sont ainsi supprimées au profit de fonctions présentes

nativement dans le langage B.

stackSize Cette variable représente la taille de la pile d'opérande *stack*. Elle peut être retirée du modèle et remplacée par un appel à la fonction *size*().

stackSizeHist Cette variable représente les différentes valeurs de la variable *stackSize* au cours du temps. Cette information peut être retrouvée en effectuant un appel à la fonction *size*() à l'index de *stackHist* souhaité.

exploredIndex Cette variable représente les différents index explorés. Les valeurs de ce vecteur sont similaire aux index de la variable instructionHist.

instruction Cette variable représente le nom de l'instruction. Il est possible de connaître cette information en lisant le nom de l'action exécutée.

La vérification de boucle est réalisée par une seule action utilisant une fonction *WHILE*. Il n'est donc pas nécessaire de conserver les instructions spécifiques *NoLoop*, *LoopEgal* et *LoopComp*. Chaque instruction est représentée par une seule action dont on sépare les pré-conditions et post-conditions dans des fichiers séparés. On utilise pour cela la clause *DEFINITIONS*. On peut ainsi réutiliser chaque spécification modélisée pour la génération et pour la vérification.
Certaines fonctions sont également isolées dans des fichiers séparés pour limiter la duplication de code.

DEFINITIONS

...

"specInstructions/smul_pre.def";

Contrairement à l'approche par Event-B, chaque instruction n'est modélisée qu'une seule fois. L'instruction *smul*, qui multiplie deux éléments de type *short* entre eux et pousse le résultat sur la pile d'opérandes, n'est par exemple représentée que par une seule opération *OP_smul* (programme 4.2) qui fait appel à 6 définitions externes (programmes 4.3, 4.4, 4.5, 4.6, 4.7 et 4.8)

La définition *smul_pre(stack)* (programme 4.3) vérifie que les préconditions nécessaires à l'exécution de l'opération sont respectées, à savoir la présence au sommet de la pile de deux variables de type *short*.

```
OP_smul =
  PRE
    smul_pre(stack)
  & common_pre(index + 1, size(stack) - 1)

  THEN
    IF (index + 1 : indexHist) THEN
      common_post_compat(generationState, index, indexHist, stack,
    stackHist, 1, offsetHist, smul, instructionHist);
      smul_post_compat(stack, stackHist(index + 1));
      generationState := loopGeneration
    ELSE
      common_post(generationState, index, indexHist, stack, stackHist,
    1, offsetHist, smul, instructionHist);
      smul_post(index, stack);
      generationState := addInstruction
    END
  END;
```

Programme 4.2 – Spécification en B de l'instruction smul

```
DEFINITIONS
  smul_pre(stack) ==
    size(stack) >= 2
  & stack(size(stack)-1) = shart
  & stack(size(stack)) = shart
```

Programme 4.3 – Préconditions de l'instruction *smul* définie dans un fichier externe

Les définitions *smul_post(index, stack)* (programme 4.4) et *smul_post_compat(stack, succ_stack)* (programme 4.5) modifient les variables d'état conformément à la post-condition de l'instruction *smul*. Les deux éléments de type *short* présents au sommet de la pile sont ainsi remplacé, soit par un élément de type *short* (pour le cas du *smul_post*), soit par un élément compatible à *short* (pour le cas du *smul_post_compat*).

Les deux définitions *common_pre* (programme 4.6) et *common_post* (programme 4.7) contiennent les pré/post-conditions communes à toutes les instructions dans un contexte linéaire ou de branchement vers une pile égale.

La définition *common_post_compat* (programme 4.8) contient les post-conditions communes à toutes les instructions dans un contexte non-linéaire avec branchement vers une pile compatible.

```
DEFINITIONS
  smul_post(index, stack) ==
    BEGIN
     index := index + 1
    ||stack := stack/|\size(stack-2) \/ {size(stack)-1 |-> shart}
    END
```

Programme 4.4 – Post-conditions de l'instruction *smul* définie dans un fichier externe

```
DEFINITIONS
  smul_post_compat(stack, succ_stack) ==
    BEGIN
      stack := stack -
        {size(stack) |-> Inference(
          {shart, succ_stack(size(succ_stack))}
        )} <+
        {size(stack)-1 |-> Inference(
          {shart, succ_stack(size(succ_stack)-1)}
        )}
    END
```

Programme 4.5 – Post-conditions de l'instruction *smul* pour le cas d'un branchement compatible

Ces trois définitions s'assurent, par exemple, du respect de l'ensemble de définition des index et des tailles de pile et effectuent le passage du système d'un état de génération à un autre en modifiant la variable *generationState*.

L'ensemble des opérations de vérification des boucles n'est ici effectué que par une seule opération *loopVerification*. Le programme 4.9 présente l'extrait de l'opération *loopVerification*. Ce programme est inspiré par l'algorithme 2 présenté en sous-section 1.3.3. On utilise une variable v initialisée à 0 que l'on utilise pour faire décroître le variant *100* - v. La valeur 100 est une valeur arbitraire choisie suffisamment grande pour permettre de vérifier la totalité de la boucle et suffisamment faible pour ne pas provoquer de timeout lors de la génération des tests par ProB. Deux conditions sont nécessaires pour sortir de la boucle de vérification : soit on est sorti de l'état *generationState=loopGeneration* en réussissant ou en échouant à la vérification de boucle, soit la variable v a dépassé la valeur 100. Une conditionnelle s'assure que la variable *stack* ne contient pas d'éléments de type Top, ce qui signifierait une incompatibilité

```
DEFINITIONS
  common_pre(post_index, post_stackSize) ==
    /* preconditions sur le generationState */
    generationState = addInstruction
    /* preconditions sur index */
  & post_index <= MaxIndex
  & post_index >= 0
    /* preconditions sur la stack */
  & post_stackSize <= MaxStackSize
  & post_stackSize >= 0
    /* preconditions sur la creation de boucle */
  & (post_index : indexHist => size(stackHist(post_index)) =
    post_stackSize)
```

Programme 4.6 – Préconditions communes à toutes les opérations du modèle

```
DEFINITIONS
  common_post(generationState, index, indexHist, stack, stackHist,
    offset, offsetHist, instruction, instructionHist) ==
    BEGIN
      indexHist := indexHist \/ {index}
    ||stackHist := stackHist \/ {index |-> stack}
    ||offsetHist := offsetHist \/ {index |-> offset}
    ||instructionHist := instructionHist \/ {index |-> instruction}
  END
```

Programme 4.7 – Post-conditions communes à toutes les opérations du modèle

```
DEFINITIONS
  common_post_compat(generationState, index, indexHist, stack, stackHist
    , offset, offsetHist, instruction, instructionHist) ==
    BEGIN
      indexHist := indexHist \/ {index}
    ||stackHist := stackHist <+ {index |-> stack}
    ||offsetHist := offsetHist \/ {index |-> offset}
    ||instructionHist := instructionHist \/ {index |-> instruction}
  END
```

Programme 4.8 – Post-conditions communes à toutes les opérations du modèle dans le cas d'un branchement vers une pile compatible

entre les types. Comme indiqué dans l'algorithme 2, la rééxecution d'une instruction lors de la vérification de type entraîne la mise à jour de la *frame stack* avec les nouvelles valeurs de piles. En début de boucle, avant l'entrée dans le *switch*, on met donc à jour la variable *stackHist* avec la valeur de la pile courante.

Chaque instruction est représentée par un cas de la clause *CASE*. Le programme 4.10 est un de ces cas. Il appelle la précondition de la définition 4.3 avec la valeur de la pile courante. Si la précondition est satisfaite alors il execute la post-condition de la définition 4.4 puis compare les valeurs d'index et de pile avec les valeurs présentes dans l'historique. Si ces valeurs sont identiques alors on a atteint un point fixe et la vérification s'arrête dans l'état *generationFinished*. Sinon, on calcule la pile compatible à la pile suivante dans l'historique et on poursuit la vérification. La vérification de boucle termine dans l'état *verificationFailed* si la pré-condition d'une instruction n'est pas respectée.

Vérification du modèle

Nous réutilisons les mêmes méthodes de vérification utilisées dans 4.2. Nous vérifions d'abord le modèle par rapport aux propriétés unitaires énoncées en partie 4.1.2. Chaque propriété est testée sur le modèle pour s'assurer que les traces valides sont bien exécutables et que les traces invalides ne le sont pas. On consigne le résultat des vérifications dans le tableau 4.9. On constate que le modèle vérifie bien l'ensemble des propriétés unitaires.

Propriété	Résultat de la vérification
EXI00 : Pas d'instruction après `return`	OK
EXI01 : Taille de piles égales lors du branchement	OK
EXI02 : Vérification de boucle si types compatibles	OK

TABLEAU 4.9 – Vérification du modèle selon les propriétés unitaires de la section 4.1.2

Nous effectuons ensuite un model-checking directement depuis l'explorateur de modèle ProB. Il ne nous a pas été possible de vérifier la totalité du modèle par model-checking après 1800 secondes.

```
PRE
   generationState = loopGeneration
THEN
  VAR v IN v := 0;
    WHILE generationState = loopGeneration & v < 100 DO

    IF Top : ran(stack) THEN
      generationState := verificationFailed
    ELSE
      stackHist := stackHist <+ {index |-> stack};

      VAR succ_index, succ_stack, tmp_stack IN
        succ_index := index + offsetHist(index);
        succ_stack := stackHist(succ_index);
        tmp_stack := stack

        CASE instructionHist(index) OF
          /* EITHER ACONST_NULL OR GOTO OR POP OR SCONST_O OR SMUL */
            ...
          ELSE
            skip
          END /* fin case */
        END /* fin switch */
      END; /* fin var while */
      v := v + 1
    INVARIANT
        v : INTEGER
      & v <= 100
    VARIANT
      100 - v
    END /* fin while */
  END /* fin var pre */
END; /* fin PRE */
```

Programme 4.9 – Opération de vérification de boucle en B classique

```
/***** SMUL *****/
OR smul THEN
  IF smul_pre(stack) THEN
    /* si precondition satisfaite --> post condition */
    smul_post(index, tmp_stack);

    IF index = succ_index
    & tmp_stack = succ_stack THEN
      /* si post = hist */
      generationState := generationFinished

    ELSE
      smul_post_compat(stack, succ_stack);
    END
  ELSE /* si precondition non satisfaite */
    generationState := verificationFailed /* sortie while */
  END
```

Programme 4.10 – Vérification de l'instruction *smul* lors de l'opération de vérification de boucle

4.3.2 Génération et concrétisation des tests fonctionnels abstraits

La génération des tests fonctionnels abstraits se fait directement depuis l'explorateur de modèle ProB. Les options à choisir dans la fenêtre *Constraint-based test case generation* sont similaires aux options choisies pour l'expérimentation Event-B.

Profondeur max.	Nb. de tests	Couverture (%)	Temps d'extract. (secondes)
1	2	29	0.70
2	2	29	2.59
3	5	71	3.01
4	6	86	6.30
5	7	100	6.10

TABLEAU 4.10 – Résultats de l'extraction des tests abstraits du modèle en B pour le prédicat *generationState = generationFinished*

Le tableau 4.10 présente les résultats de l'extraction des tests abstraits à partir du modèle en B. On constate qu'il est possible de générer rapidement des tests couvrant toutes les instructions. Cependant, il ne nous est pas possible de nous assurer

à ce stade que ces instructions déclencheront une vérification de boucle et seront rééxécutés lors de la vérification. On met donc en place sur le modèle une variable booléenne *instructionVerified* initialisée à *FALSE* que l'on passe à *TRUE* une fois la post-condition de l'instruction choisie exécutée. On obtient ainsi le comportement *GetStack* de l'expérimentation précédente. On réalise l'extraction avec le prédicat *generationState = generationFinished & instructionVerified=TRUE* autant de fois qu'il y a d'instruction à couvrir. La profondeur maximale de recherche est fixée à 7.

Instruction ciblée	Temps d'extraction (sec.)	Nombre de tests
aconst_null	3077.270	0
bspush	3021.330	0
goto	1791.480	2
pop	1850.110	2
sconst_0	3199.610	0
smul	2602.390	1

TABLEAU 4.11 – Extraction des tests abstraits du modèle en B pour le prédicat *generationState = verificationFinished & instructionVerified = TRUE* pour chaque instruction

Le tableau 4.11 présente les résultats de la seconde extraction de tests abstraits à partir du modèle en B. On constate qu'il n'est pas possible de générer des tests dans lesquels les instructions *aconst_null*, *bspush* et *sconst_0* déclenchent une vérification de types.

La génération des tests abstraits fonctionnels utilise le même outil que pour l'expérimentation précédente. Pour concrétiser ces tests nous pouvons donc réutiliser les outils développés dans 4.2.2 et 4.2.3.

A l'issue de ces étapes, 12 tests sont générés et concrétisés au format CAP.

4.3.3 Exécution des tests fonctionnels sur le VCI

L'exécution des tests fonctionnels sur le VCI réutilise les outils utilisés dans 4.2.4.

Le tableau 4.12 présente les résultats de la vérification par le VCI des tests concrets fonctionnels. Les 12 tests exécutés ont été acceptés par le VCI.

Identifiant	Trace	VCI
CAP01	goto 0	OK
CAP02	return	OK
CAP03	aconst_null, goto 0	OK
CAP04	sconst_0, goto 0	OK
CAP05	bspush, goto 0	OK
CAP06	aconst_null, pop, goto 0	OK
CAP07	sconst_0, sconst_0, smul, goto 0	OK
CAP08	aconst_null, goto 3, goto -2, sconst_0, pop	OK
CAP09	aconst_null, goto 3, goto -2, bspush, pop	OK
CAP10	sconst_0, pop, bspush, goto -2	OK
CAP11	aconst_null, sconst_0, pop, bspush, goto -2	OK
CAP12	bspush, sconst_0, sconst_0, smul, sconst_0, goto -2	OK

TABLEAU 4.12 – Résultats de la vérification par le VCI des tests concrets fonctionnels

4.3.4 Génération semi-automatique des modèles mutants

Le VTG n'est capable de générer des mutations à partir de modèles Event-B uniquement. Son adaptation pour le B classique nécessiterait plusieurs mois.

Les modèles B peuvent être modifiés à l'aide d'un éditeur de texte. Il nous est donc possible de faire faire la génération des mutant par un script. Ce script, dont le fonctionnement est détaillé en section 5.3, génère des modèles mutants à partir d'un modèle "patron" dans lequel on a prédéfini un point d'injection pour les spécifications mutées.

Pour générer les modèles mutants, nous suivons le même protocole que pour la précédente expérimentation. Nous appliquons ce protocole à la mutation de l'instruction *smul* lors de la vérification de boucle.

La différence par rapport à l'expérimentation précédente est qu'ici les types en précondition de l'instruction *smul* sont modifiés par un script.

DEFINITIONS

$$\texttt{smul\_pre\_MUT\_ID}(\texttt{stack}) ==$$
$$\texttt{size}(\texttt{stack}) >= 2$$
$$\&\texttt{stack}(\texttt{size}(\texttt{stack}) - 1) = \texttt{TYPE1}$$
$$\&\texttt{stack}(\texttt{size}(\texttt{stack})) = \texttt{TYPE2}$$

Dans cette expérimentation la spécification de chaque instruction est divisée en trois fichiers appelés dans la clause *DEFINITIONS*. On souhaite muter la précondition de smul. Pour chaque mutant généré, on créera un nouveau fichier contenant les gardes mutés que l'on ajoutera dans la clause *DEFINITIONS*.

```
DEFINITIONS

    "specInstructions/smul__pre__MUT__ID.def";

    "specInstructions/smul__pre.def";

    "specInstructions/smul__post.def";

    "specInstructions/smul__post__compat.def";
```

Chaque instruction mutée est intégrée dans un modèle muté séparé et identifié par son identifiant *MUT_ID*.

4.3.5 Génération et concrétisation des tests de vulnérabilité abstraits

Pour générer les tests de vulnérabilité abstraits nous réutilisons les outils développés dans 4.2.2 et 4.2.3.

Pour couvrir l'instruction smul dans le cas de sa vérification par le VCI, on utilise le protocole suivi pour l'extraction de tests fonctionnels. On utilise une variable booléenne *instructionVerified* passée à *TRUE* lors de la réexecution de l'instruction *smul*. Le prédicat utilisé pour le CBC est donc le même que pour la génération fonctionnelle, à savoir $generationState = generationFinished \& instructionVerified = TRUE$.

On réalise la procédure d'extraction sur chacun des modèles mutants en fixant la profondeur maximale à 7.

Modèle	Nombre de tests	Temps d'extraction (sec.)
MUT00	0	1958.030
MUT01	0	1959.110
MUT02	0	1976.420
MUT03	0	1956.880
MUT04	1	364.980
MUT05	0	1996.650

TABLEAU 4.13 – Génération des tests de vulnérabilité concrets

Le tableau 4.13 présente le nombre de tests de vulnérabilité abstraits générés à partir de chacun des modèles mutants. On constate que les modèles mutants *MUT00*, *MUT01*, *MUT02*, *MUT03* et *MUT05* n'ont pas permis de générer de tests. Cela signifie qu'il n'est pas possible au CBC de trouver de traces répondant aux contraintes données.

Le test de vulnérabilité abstrait obtenu par l'exploration du modèle mutant *MUT04* est le suivant : *{sconst_0, sconst_0, smul, bspush, goto -2}*.

La définition de la précondition ayant mené à ce test est la suivante :

DEFINITIONS

$$\texttt{smul\_pre\_MUT04}(\texttt{stack}) ==$$
$$\texttt{size}(\texttt{stack}) >= 2$$
$$\&\texttt{stack}(\texttt{size}(\texttt{stack}) - 1) = \texttt{shart}$$
$$\&\texttt{stack}(\texttt{size}(\texttt{stack})) = \texttt{numericTypes}$$

On concrétise ce test de vulnérabilité abstrait en réutilisant l'outil développé en sous-section 4.2.3.

4.3.6 Exécution des tests de vulnérabilité concrets sur le VCI

Pour exécuter le test de vulnérabilité concret sur le VCI nous réutilisons l'outil développé pour 4.2.4.

Identifiant	Trace	VCI
MUT04_CAP00	sconst_0, sconst_0, smul, bspush, goto -2	OK

TABLEAU 4.14 – Vérification par le VCI du test de vulnérabilité concret

Le tableau 4.14 présente les résultats de la vérification par le VCI du test de vulnérabilité extrait lors de l'étape précédente. On constate que le test de vulnérabilité soumis au VCI lors de cette phases est accepté par le VCI. Contrairement à ce que spécifie la spécification Java Card de l'instruction *smul*, il semble que le VCI accepte de multiplier un *short* avec un *byte*. Cela ne constitue pas une vulnérabilité du VCI car, en réalité, les *bytes* et les *short* sont stockés sur les mêmes emplacements mémoire sur la pile. On parlera donc plutôt ici d'incohérence dans la spécification Java Card.

4.3.7 Statistiques et observations

Cette expérimentation nous a permis d'étudier les avantages et inconvénients du passage de notre modèle en B classique. La phase de réalisation du modèle B a nécessité 4 mois de travail entre septembre et décembre 2017. Elle a été sensiblement plus rapide que celle du modèle Event-B pour deux raisons. Le cas d'étude, similaire à celui de l'expérimentation précédente, n'a pas nécessité d'étude complémentaire pour être transposé au langage B. Ensuite, la réduction du nombre de variables et d'opérations et l'utilisation de fonctions natives de ProB a permis de raccourcir le temps de réalisation du modèle. L'utilisation de la clause *DEFINITIONS* nous a également permis de diminuer la duplication de code. La machine principale contient 8 opérations, dont 7 pour les instructions et 1 pour l'étape de vérification de boucle.

La génération des tests fonctionnels a débuté en septembre 2018. L'approche que nous avons adoptée pour cette expérimentation nous a contraint à repenser nos outils et nos méthodes de génération de test. L'outil de concrétisation développé dans l'expérimentation précédente a ainsi dû être retravaillé pour s'adapter à la nouvelle structure des tests abstraits XML. Aucune adaptation n'a en revanche été nécessaire sur l'outil d'exécution des tests. Nous avons généré les tests en deux temps pour couvrir dans un premier temps le contexte linéaire et dans un second temps le contexte non-linéaire.

La réalisation des mutations à partir du modèle fonctionnel a débuté en janvier 2019. Il nous a été possible d'utiliser un script pour automatiser la génération des modèles mutants. Nous avons ainsi réalisé 6 modèles mutants en moins d'une seconde. La génération des tests de vulnérabilité à partir de ces modèles mutants a duré environs 3 heures. La structure des modèles mutants ne diffère pas de celle du modèle fonctionnel. Aucune adaptation des outils de concrétisation et d'exécution n'a donc été requise.

4.4 Comparaison des expérimentations

Nos expérimentations nous permettent d'établir en premier lieu un comparatif entre B et Event-B. L'expérimentation Event-B a abouti à la création de deux modèles : l'un de 30 instructions et l'autre de 7. L'expérimentation B s'est basée sur le second modèle. Dans cette section, nous comparons nos expérimentations selon des critères de performance, d'adaptabilité dans un modèle complet et de coût en temps humain.

4.4.1 Comparaison des performances

Le modèle Event-B complet de 30 instructions nous a permis de couvrir une grande quantité d'instructions Java Card. Cependant, le temps nécessaire pour en extraire des traces nous a contraint à créer un second modèle plus léger. Sur ce second modèle limité à 7 instructions, il nous a été possible d'extraire des tests de longueur 7 et de couvrir 100% des instructions dans un contexte linéaire, non-linéaire, avec et sans vérification de types.

Le modèle B, basé sur ce second modèle, nous a permis d'atteindre les objectifs en contexte linéaire en un temps plus réduit. L'utilisation de la clause *WHILE* nous a permis de réaliser toutes les étapes de vérification en une seule opération. La réalisation des objectifs en contexte non-linéaire en revanche a nécessité des modifications au modèle ainsi qu'une génération indépendante pour chaque instruction, ce qui a allongé le temps de génération global.

Critère de comparaison	Modèle Event-B	Modèle B
Nombre de tests fonctionnels générés	24	7
Temps d'extraction des tests fonctionnels (sec.)	1812	15546
Nb. de tests fonctionnels acceptés par le VCI	23/24	7/7
Nombre de tests de vulnérabilité générés	1	1
Temps d'extraction des tests de vulnérabilité (sec.)	7815	10209
Nb. de tests de vulnérabilité acceptés par le VCI	0/1	1/1

TABLEAU 4.15 – Comparatif des performances réalisées par les deux expérimentations

4.4.2 Comparaison d'adaptabilité dans un modèle complet

L'approche par Event-B est celle qui permet la plus grande adaptabilité dans un modèle complet. L'approche par raffinement et l'utilisation de conventions de nommage nous permet de l'intégrer facilement dans le modèle proposé par Savary.

Dans l'approche par B en revanche, chaque action est modélisée dans une unique couche. Bien que l'utilisation des définitions nous permette une meilleure réutilisation du code, cette approche nécessiterait beaucoup de modifications pour être intégrée à un modèle complet.

4.4.3 Comparaison de coût en temps humain

Dans l'approche par Event-B, chaque instruction est représentée 4 fois selon son contexte *NoLoop*, *LoopEgal*, *LoopCompatible* ou *GetStack*, car le langage Event-B ne contient pas de clauses $IF...THEN...ELSE$ ni de $WHILE$. Il faut donc créer un événement pour chaque cas. Le temps nécessaire pour modéliser chaque instruction est donc plus long comparé à l'approche en B. Chaque modification à la spécification d'une instruction doit également être répliquée 4 fois, ce qui allonge le temps nécessaire à l'évolution du modèle. De plus, l'outil Rodin souffre de nombreuses lenteurs, ce qui allonge encore plus le temps de développement.

L'approche B présente ici un avantage important sur l'approche Event-B. Le temps nécessaire pour faire évoluer le modèle est très largement réduit grâce à l'utilisation de la clause *DEFINITIONS*. Nous avons pu réaliser le modèle B en seulement 4 mois

car nous avions déjà une bonne compréhension du cas d'étude.

Chapitre 5

Outils développés

Un des objectifs de ce travail de recherche est l'amélioration de la chaîne d'outils développée par Savary et al. dans [15].

La première section de ce chapitre présente les scripts utilisés pour réaliser certaines parties des modèles. Dans une seconde section, nous décrivons l'outil de concrétisation développé pour générer des tests au format JCA et les convertir au format CAP. Cet outil dispose également d'une fonction d'exécution permettant de tester le fichier CAP sur le VCI.

La dernière section présente l'outil de génération semi-automatique de modèles mutants utilisé dans l'expérimentation en B.

5.1 Disjonction des ensembles *Instructions* et *GenerationStates*

Les modèles réalisés dans les expérimentations du chapitre 4 utilisent deux ensembles *Instructions* et *GenerationStates* représentant respectivement l'ensemble des instructions disponibles et les différents états du système. Pour que notre modèle Event-B soit en mesure de différencier les éléments de ces ensembles, il est nécessaire d'écrire un prédicat contenant la disjonction de tous les éléments de chaque ensemble.

Le programme 5.1 présente le résultat de la disjonction de l'ensemble $STATES = \{addInstruction, generationFinished, loopGeneration, verificationFailed\}$

```
addInstruction /= generationFinished & addInstruction /= loopGeneration
    & addInstruction /= verificationFailed & generationFinished /=
    loopGeneration & generationFinished /= verificationFailed &
    loopGeneration /= verificationFailed
```

Programme 5.1 – Disjonction des élements de l'ensemble *STATES*

On réutilise ce même outil pour générer la disjonction de l'ensemble des instructions.

5.2 Concrétisation des tests fonctionnels abstraits

Les tests extraits par la fonction CBC de ProB sont au format XML. Le VCI n'accepte que des tests au format CAP. Nous réalisons donc un outil en trois parties capable de faire passer les tests d'un format abstrait au format concret, que l'on exécute ensuite sur le VCI.

5.2.1 Lecture de la suite de tests abstraits XML

Nous développons tout d'abord un «*parser*» de fichiers XML en Python. Cet extracteur de tests est capable de s'adapter aux structures de tests en Event-B et en B. En effet, en Event-B les opérations représentant les instructions ont une structure complexe, sous la forme *Numero de raffinement + type d'instruction + instruction* tandis qu'en B les instructions ne sont préfixés que par *OP_*

Le programme 5.2 présente un extrait d'une suite de test au format XML pour le modèle Event-B. La trace extraite ici est *[aconst_null, pop, goto -2]*. La lecture de l'*offset* du goto se fait en lisant la valeur de sa balise `<modified name="offset">` dont il est le parent.

5.2.2 Concrétisation des tests abstraits vers le format JCA

La trace abstraite extraite du fichier XML est ensuite concrétisée au format JCA pour pouvoir être soumise au VCI. On crée pour cela le programme java 5.3 qui contient une méthode vide.

```
<extended_test_suite>
    <test_case>
        <initialisation >...</initialisation>
        <step name="R15_aconst_null">
            ...
        </step>
        <step name="R15_pop">
            ...
        </step>
        <step name="R15_goto">
            ...
            <modified name="offset">-2</modified>
            ...
        </step>
    </test_case>
</extended_test_suite>
```

Programme 5.2 – Structure détaillée d'une suite de tests au format XML pour l'Event-B

```
private static void test()
{
}
```

Programme 5.3 – Méthode *test()* vide en Java

Une fois ce fichier Java compilé on obtient un fichier *.class*. Ce fichier *.class* est ensuite converti en utilisant la librairie *com.sun.javacard.converter.Converter*. Une fois converti on obtient un fichier *.cap* et son équivalent au format *.jca*.

Le programme 5.4 représente l'implémentation JCA de la méthode `test()`.

Nous en déduisons que la charge active du test doit être placée à la place de la ligne `L0: return;`. Les offset des goto sont remplacés par une valeur d'étiquette *Lx* à déterminer. La valeur de *.stack* permet de fixer la taille maximale de la pile d'opérandes. Par défaut, cette valeur est à 0. Il faut donc la remplacer par la valeur de *MaxStackSize* choisie dans les modèles.

```
.method private static test()V
{
.stack 0;
.locals 0;

L0: return;
}
```

Programme 5.4 – Méthode *test()* vide en JCA

Programme 5.5 – Test abstrait

```
sconst_0;
sconst_0;
goto -1
```

Programme 5.6 – Test concret JCA

```
L0:     sconst_0;
L1:     sconst_0;
        goto L1;
```

5.2.3 Conversion des fichiers JCA en fichiers CAP et vérification par le VCI

La conversion des fichiers JCA en fichiers CAP utilise la librairie *capgen* de Java Card. Chaque fichier CAP obtenu est ensuite vérifié à l'aide de la librairie *offcardverifier* de Java Card. En cas d'échec le VCI renvoie la raison du rejet dans la console, comme visible en programme 5.7.

```
Verification du fichier CAP output/test1.cap
Erreur: Dans la methode Descriptor[67]/Method[69]:
Instruction au PC 3: Hauteurs de pile incoherentes a un point de
   jointure (0 / 1 elements)
La verification est terminee, 0 avertissements et 1 erreur.
```

Programme 5.7 – Erreur affichée dans la console par le VCI

5.3 Génération semi-automatique de modèles mutants

A la différence des modèles Event-B édités depuis l'outil Rodin, les modèles B ont l'avantage de pouvoir être modifiés depuis un éditeur de texte traditionnel. Il nous est donc possible d'utiliser un script Python pour la génération des modèles mutants de manière semi-automatique.

Pour cela, on prend la précondition de l'instruction à muter, dans laquelle on ajoute le suffixe *MUT_ID*. On remplace ensuite le ou les types en précondition de cette instruction par le texte *TYPE* suivi d'un numéro. Par exemple, la précondition avant mutation de l'instruction *smul* devient la suivante :

DEFINITIONS

smul_pre_MUT_ID(stack) ==
size(stack) >= 2
&stack(size(stack) − 1) = TYPE1
&stack(size(stack)) = TYPE2

On prépare également le modèle en ajoutant la ligne correspondant à la précondition de l'instruction à muter dans les définitions et on modifie les appels à celle-ci dans les emplacements correspondants du modèle. Les définitions du modèle contenant, par exemple, la préconditions de l'instruction *smul* à muter deviennent les suivantes :

DEFINITIONS

"specInstructions/smul_pre_MUT_ID.def";
"specInstructions/smul_pre.def";
"specInstructions/smul_post.def";
"specInstructions/smul_post_compat.def";

Les appels à *smul_pre_(stack)* sont remplacées par smul_pre_MUT_ID(stack) aux emplacements adéquats dans le modèle.

Le script est ensuite capable, une fois le modèle préparé en suivant le protocole décrit ci-dessus, de générer toutes les combinaisons possibles dans un ensemble de types donné puis de créer les fichiers de définitions à injecter dans des modèles distincts, suffixés par leur identifiant de mutation *MUT_ID*.

Conclusion

Les précédentes recherches entreprises sur la vérification du VCI par la méthode VTG ont permis de générer des tests de vulnérabilité pour des programmes Java Card séquentiels. L'Event-B était alors l'unique méthode formelle utilisée pour générer ces tests. Notre travail de recherche s'est donc porté sur l'extension des précédents travaux au cas des programmes itératifs. Nous avons proposé une réponse à ce problème en permettant également un comparatif entre l'Event-B au B.

La première réponse apportée à notre problème initial a utilisé la méthode formelle Event-B. Cette première approche nous a permis de repartir d'une base existante et de l'adapter au cas des programmes itératifs. L'approche par raffinement nous a permis d'aborder le problème couche par couche, de la plus abstraite à la plus concrète. Cette méthode a trouvé ses limites lorsqu'il a fallu limiter la taille des traces générées. Nous avons donc réalisé un second modèle en B, et tiré profit de la présence de l'instruction *while* du langage B pour réduire la taille des traces de vérification à une seule opération. De plus, lors de l'approche par la méthode B, il a été possible de réaliser des manipulations de nos machines B par des scripts et ainsi générer automatiquement des modèles mutants. Un des modèles mutants nous a permis d'extraire un test de vulnérabilité qui a été accepté par le VCI.

L'extraction des tests produits par le modèle en Event-B nous a permis de couvrir en un temps raisonnable l'ensemble de nos objectifs. Cependant, une erreur dans la modélisation des branchements compatibles a été mise en lumière lors de l'extraction des tests concrets. Nous l'avons corrigée lors de l'expérimentation en B en durcissant les gardes relatives aux boucles compatibles et avons obtenu des temps de génération de tests séquentiels très satisfaisants. Cependant, l'approche choisie pour l'expérimentation en B a nécessité de légères modifications pour atteindre nos objectifs de tests,

ce qui a allongé le temps de génération des tests.

Pour pouvoir apporter une réponse complète au problème de la comparaison entre l'Event-B et le B, il serait intéressant de corriger les erreurs du modèle Event-B en durcissant les gardes relatives à la génération de boucles compatibles. Il serait ensuite possible d'automatiser la génération des modèles mutants en apportant les modifications nécessaires au VTG pour utiliser la nouvelle API ProB 2.0. Toutefois, le temps requis pour générer des tests demeurera toujours plus important en Event-B qu'en B du fait de l'absence d'instruction *while* dans le langage Event-B. L'approche par B nous semble donc plus prometteuse pour l'évolution future de la modélisation du VCI.

Concernant ces évolutions, nous pouvons envisager plusieurs pistes pour répondre aux différentes limitations de notre modèle. La constante MaxIndex, fixée à 10 pour notre expérimentation, pourrait être étendue à des valeurs supérieures pour permettre la génération de programmes plus complexes, comme des programmes contenant des boucles doubles ou multiples. Il est cependant probable que l'ajout d'instructions comme if, else if, else et switch provoque une explosion combinatoire compte tenu de la taille importante de l'espace d'états. Une piste alternative d'amélioration du modèle pourrait alors être de conserver la limite des boucles simples mais d'ajouter au modèle les variables locales et les instructions capables de les manipuler.

Annexe A

Équivalences entre code Java et code JCA

A.1 Méthode vide

Programme A.1 – Code Java

```
// methode vide
```

Programme A.2 – Code JCA

```
L0:     return;
```

A.2 Affectation de variable

Programme A.3 – Code Java

```
short variable = 5;
```

Programme A.4 – Code JCA

```
L0:     sconst_5;
        sstore_0;
        return;
```

A.3 Affectation de variable et incrémentation

Programme A.5 – Code Java

```
short variable = 0;
variable += 10;
```

Programme A.6 – Code JCA

```
L0:     sconst_0;
        sstore_0;
        sinc 0 10;
        return;
```

A.4 Boucle infinie

Programme A.7 – Code Java

```
while (true);
```

Programme A.8 – Code JCA

```
L0:     goto L0;
```

A.5 Affectation et incrémentation infinie

Programme A.9 – Code Java

```
short variable = 0;
while(true) {
        variable++;
}
```

Programme A.10 – Code JCA

```
L0:     sconst_0;
        sstore_0;
L1:     sinc 0 1;
        goto L1;
```

A.6 Affectation, conditionnelle et incrémentation

Programme A.11 – Code Java

```
short variable = 0;
if (variable != 0) {
        variable++;
}
```

Programme A.12 – Code JCA

```
L0:     sconst_0;
        sstore_0;
        sload_0;
        ifeq L2;
L1:     sinc 0 1;
L2:     return;
```

A.7 Boucle finie (for) et incrémentation

Programme A.13 – Code Java

```
for (short i = 0; i <= 5; i++) {
        i += 2;
}
```

Programme A.14 – Code JCA

```
L0:     sconst_0;
        sstore_0;
L1:     sload_0;
        sconst_5;
        if_scmpgt L3;
L2:     sinc 0 2;
        sinc 0 1;
        goto L1;
L3:     return;
```

A.8 Boucle finie (switch) et affectation

Programme A.15 – Code Java

```
short variable = 0;
switch(variable) {
        case 2: variable = 2;
        break;
        default: variable = 4;
        break;
}
```

Programme A.16 – Code JCA

```
L0:     sconst_0;
        sstore_0;
        sload_0;
        slookupswitch L2 1 2 L1;
L1:     sconst_2;
        sstore_0;
        goto L3;
L2:     sconst_4;
        sstore_0;
L3:     return;
```

Bibliographie

[1] Hiralal Agrawal, Richard DeMillo, R_ Hathaway, William Hsu, Wynne Hsu, Edward W Krauser, Rhonda J Martin, Aditya P Mathur et Eugene Spafford.
« Design of mutant operators for the C programming language ».
Rapport Technique, Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue . . . , 1989.

[2] Jean-Louis Lanet Aymerick Savary, Marc Frappier.
« Detecting Vulnerabilities in Java-Card Bytecode Verifiers using Model-Based Testing ».
Lecture Notes in Computer Science, 7940:223–237, 2013.

[3] Jean-Louis Lanet Aymerick Savary, Marc Frappier, Michael Leuschel.
« Model-Based Robustness Testing in Event-B using Mutation ».
Software Engineering and Formal Methods, 9276:132–147, 2015.

[4] Jordan Bouyat et Fernand Lone-Sang.
« Recherche de vulnérabilités dans les piles USB : approches et outils ».
Actes du Symposium sur la Sécurité des Technologies de l'Information et de la Communication (SSTIC) 2014, 2014.

[5] Andrea Calvagna et Emiliano Tramontana.
« Combinatorial Validation Testing of Java Card Byte Code Verifiers ».
2013 Workshops on Enabling Technologies : Infrastructure for Collaborative Enterprises, pages 347–352, 2013.

[6] Michael Felderer, Philipp Zech, Ruth Breu, Matthias Büchler et Alexander Pretschner.

« Model-based security testing : a taxonomy and systematic classification ».
Software Testing, Verification and Reliability, 26(2):119–148, 2016.

[7] Marc Frappier.
« IGL501 - Méthodes formelles en génie logiciel ».
Notes de cours, Université de Sherbrooke.

[8] Henri Habrias et Marc Frappier.
Software specification methods.
Wiley Online Library, 2006.

[9] Sunwoo Kim, John Clark et John McDermid.
« The rigorous generation of Java mutation operators using HAZOP ».
Informe técnico, The University of York, 1999.

[10] Kim N King et A Jefferson Offutt.
« A fortran language system for mutation-based software testing ».
Software : Practice and Experience, 21(7):685–718, 1991.

[11] Luc Lavoie.
« IGL 601 - Techniques et outils de développement ».
Notes de cours, Université de Sherbrooke.

[12] Sun Microsystems.
« Virtual machine specification Java Card platform ».
http ://www.oracle.com.

[13] Collin Mulliner et Charlie Miller.
« Fuzzing the Phone in your Phone ».
Black Hat USA, 25:31, 2009.

[14] A Jefferson Offutt, Jeff Voas et Jeff Payne.
« Mutation operators for Ada ».
Rapport Technique, Technical Report ISSE-TR-96-09, Information and Software Systems Engineering . . . , 1996.

[15] Aymerick Savary.
« *Détection de vulnérabilités dans la vérification de code intermédiaire de Java Card* ».
Thèse de doctorat, Université de Sherbrooke et Université de Limoges, 2015.

[16] Ari Takanen, Jared D Demott, Charles Miller et Atte Kettunen.
Fuzzing for software security testing and quality assurance.
Artech House, 2018.

[17] Mark Utting et Bruno Legeard.
Practical model-based testing : a tools approach.
Elsevier, 2010.

[18] J. Woodcock et J. Davies.
Using Z : Specification, Refinement, and Proof.
Prentice-Hall international series in computer science. Prentice Hall, 1996.